MODULO SCHEDULING LOOPS ONTO COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

RANI GNANAOLIVU

Modulo Scheduling Loops onto Coarse-Grained Reconfigurable Architectures

by Rani Gnanaolivu

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

January 2013

St. John's Newfoundland

Abstract

Reconfigurable systems have drawn increasing attention from both academic researchers and creators of commercial applications in the past few years because they could combine flexibility with efficiency. There are two main types of reconfigurable architectures – fine-grained and coarse-grained. The functionality of fine-grained architecture hardware is specified at the bit level while the functionality of the coarse-grained architecture hardware is specified at the word level. Coarse-grained reconfigurable architectures (CGRAs) have gained currency in recent years due to their abundant parallelism, high computational intensity and flexibility. A CGRA normally is comprised of an array of basic computational and storage resources, which are capable of processing a large volume of applications simultaneously. To exploit the inherent parallelism in the applications to enhance performance, CGRAs have been structured for accelerating computation intensive parts such as loops, that require large amounts of execution time. The loop body is essentially drawn onto the CGRA mesh, subject to modulo resource usage constraints. Much research has been done to exploit the potential parallelism of CGRAs to increase the performance of time-consuming loops. However, sparse connectivity and distributed register files present difficult challenges to the scheduling phase of the CGRA compilation framework. While traditional schedulers do not take routability into consideration, software pipelining can improve the scheduling of instructions in loops by overlapping instructions from different iterations. Modulo scheduling is an approach for constructing software pipelines that focuses on minimizing the time between the initiations of iterations — the so-called initiation interval (II). For example, if a new iteration is

started every II cycles, the time to complete n iterations will approach $II \times n$, for large n loops, thereby maximizing performance.

The problems of scheduling (*deciding when an operation should happen*), placing (*deciding where an operation should happen*), and routing (*the problem of how information travels though space and time between operations*) can be unified if they are modelled by a graph embedding problem. The data flow graph of the loop is embedded in a routing resource graph representing the hardware across a number of cycles equal to the initiation interval.

Particle swarm optimization (PSO) has shown to be successful in many applications in continuous optimization problems. In this thesis, we have proposed algorithms to solve scheduling, placing, and routing of loop operations simultaneously by using PSO. We call this approach modulo-constrained hybrid particle swarm optimization (MCHPSO). There are many constraints and one optimization objective, which is the II that needs to be considered during the mapping and scheduling procedure. The scheduling algorithm tries to minimize the initiation interval to start the next iteration of the loop under the resource and modulo constraints for the architecture being used.

When conditional branches such as if-then-else statements are present in the loop, they create multiple execution paths. Exploiting conditional branches through our predicated exclusivity, the MCHPSO algorithm reuses the resources which are in the exclusive execution paths and which may allow the loop to be scheduled with a lower II. Finally, a priority scheme algorithm along with recurrence aware modulo scheduling is proposed to map inter-iteration dependencies onto CGRAs, which is able to save resources for all recurrences cycles and to map remaining operations.

Acknowledgements

First and foremost I would like to thank God for the wisdom and perseverance that he has blessed me with during this PhD program, and indeed, throughout my life: "He who began a good work in you will carry it on to completion until the day of Christ Jesus." (Philippians 1: 6)

It is my pleasure to thank many people who made this thesis possible. I express my sincere thanks to my supervisors, Dr. T. S. Norvell and Dr. R. Venkatesan, for their intellectual assistance, financial support, and continuous encouragement during my research. Their enthusiasm, inspiration and sound advice was motivational and helped me through even the roughest patches of my graduate program. I thank Dr. P. Gillard for taking time to read my work and offer invaluable comments and suggestions. I thank NSERC for supporting my research at Memorial. I thank Shuang Wu for teaching me his work in generating data flow graph from a HARPOL program. I thank him for allowing me to use his work for my test cases in the PhD program.

Last but not the least; I thank my family for their boundless love, encouragement, and unconditional support, both financially and emotionally throughout my PhD program. Especially, I thank Mohan Gnanaolivu, my father in-law, for his valuable editorial corrections. I also thank my loving, supportive, husband Praveen Gnanaolivu whose faithful support during the final stages of this PhD is so appreciated. I would also like to thank my loving son Kevin for the sincere everyday prayers for the completion of my research. I thank all my friends for constantly encouraging me and reminding me of my aspirations.

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

CGRA	Coarse Grained Reconfigurable Architecture
FPGA	Field Programmable Gate Array
II	Initiation Interval
MII	Minimal Initiation Interval
DFG	Data Flow Graph
TA	Target Architecture
PSO	Particle Swarm Optimization
MCHPSO	Modulo Constrained Hybrid Particle Swarm Optimization
RRG	Routing Resource Graph
HARPO/L	HARdware Parallel Objects Language
MRT	Modulo Reservation Table
ASAP	As Soon As Possible
ALAP	As Late As Possible
DRESC	Dynamically Reconfigurable Embedded Systems Compiler
MRRG	Modulo Routing Resource Graph
FU	Functional Unit
RF	Register File

CB	Column Bus
RB	Row Bus
SRF	Shared Register File
IPC	Instruction Per Cycle
MU	Memory Unit
VLIW	Very Long Instruction Word
DSP	Digital Signal Processing
ASIC	Application Specific Integrated Circuit
SA	Simulated Annealing
ACO	Ant Colony Optimization
GA	Genetic Algorithm
ILP	Instruction Level Parallelism
TLP	Task Level Parallelism

Chapter 0

Introduction

0.0 Reconfigurable Computing

Reconfigurable systems [Abielmona, 2009] have drawn increasing attention from both academic and commercial researchers in the past few years because they combine flexibility with efficiency and upgradability [Todman *et al.*, 2005]. The flexibility in reconfigurable devices mainly comes from their routing interconnect. Reconfigurable computing fills the gap between application-specific integrated circuits (*ASICs*) and general purpose processors (*GPPs*), as described in Figure 0.0. When compared with GPPs, reconfigurable computing has the ability to make substantial changes in the data path, in addition to the control flow. However, when compared with ASICs, it has the possibility to adapt the hardware during the runtime by "loading" a new configuration in the memory. To avoid the bandwidth limitation between processor and memory, called the Von Neumann bottleneck, a portion of the application is mapped directly onto the hardware to increase the data parallelism in reconfigurable

computing.

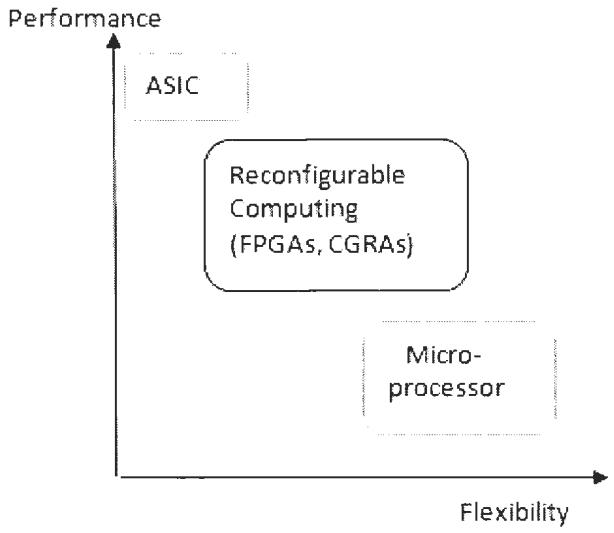


Figure 0.0: Advantages of Reconfigurable Computing

The principal benefits of reconfigurable computing compared with ASICs and GPPs are the ability to design larger hardware with fewer gates and to realize the flexibility of a software-based solution while retaining the execution speed of a more traditional, hardware-based approach [Barr, 1998]. Due to the dynamic nature of reconfigurable computing, it is advantageous to have the software manage the process of deciding which hardware objects to execute.

Reconfigurable architectures are broadly classified into fine-grained and coarse-grained. The first devices that had been used for fine-grained reconfigurable computing were the field-programmable gate arrays (*FPGAs*). An FPGA consists of a matrix of programmable logic cells, executing bit-level operations, with a grid of interconnect lines running among them. FPGAs allow realizing systems from a low

granularity level, that is, logic gates and flip-flops. This makes FPGAs very popular for the implementation of complex bit level operations. However, FPGAs are inefficient for coarse-grained data path operations due to the high cost of reconfiguration performance and power [Hartenstein, 2001]. The coarser granularity greatly reduces the delay, power and configuration time relative to an FPGA device at the expense of reduced flexibility [Dimitroulakos *et al.*, 2007]. However, coarse-grained reconfigurability has the advantage of much higher computational density compared to the FPGAs.

0.1 Coarse-Grained Reconfigurable Architecture

Coarse-grained reconfigurable architectures (*CGRAs*) have been emerging as a potential candidate for embedded systems in recent years. CGRAs have a data-path of word width whereas fine-grained architectures are much less efficient and have huge routing area overhead and poor routability. A major benefit of CGRAs over FPGAs is a massive reduction of configuration memory, configuration time, and complexity reduction of the Placement and Routing (*P&R)* problem [Hartenstein, 2001]. These architectures combine with the high performance of ASICs and the flexibility of microprocessors, to accelerate computation intensive parts of applications in embedded systems [Dimitroulakos *et al.*, 2007]. However, there are still many outstanding issues such as a lack of a good design methodology to exploit high performance and efficiency on CGRAs [Vassiliadis and Soudris, 2007a].

CGRAs consist of programmable, hardwired, coarse-grained processing elements (*PEs*), which support a predefined set of word-level operations while the intercon-

nection network is based on the needs of a specific architecture domain. A generic architecture of a coarse-grain reconfigurable system, shown in Figure 0.1, encompasses a set of coarse-grain reconfigurable units (*CGRUs*), a programmable interconnection network, a configuration memory, and a controller. The coarse-grained reconfigurable array executes the computationally-intensive parts of the application while the main processor is responsible for the remaining parts of the application.

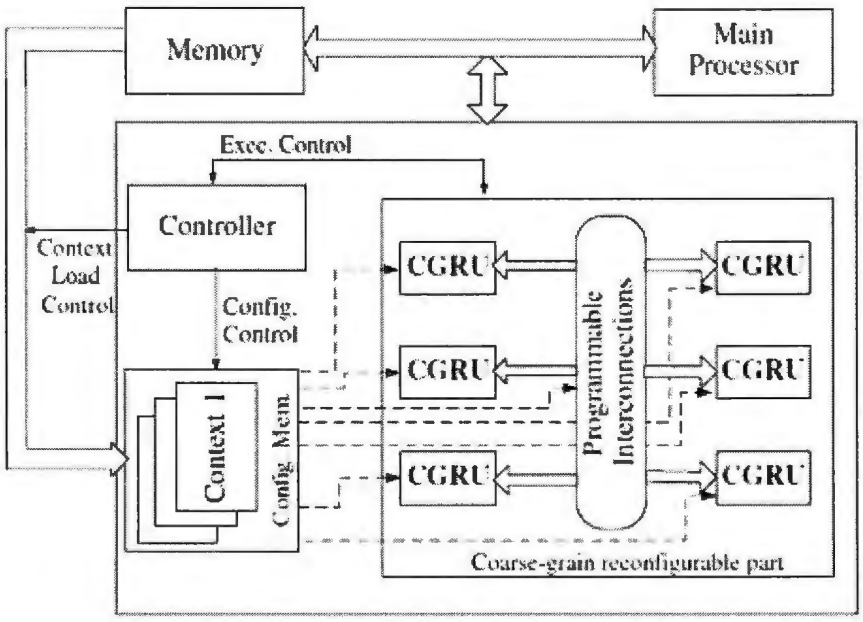


Figure 0.1: A Generic Coarse-Grain Reconfigurable System taken from [Vassiliadis and Soudris, 2007a].

The domain-specific, hardwired, CGRU executes a logical or arithmetic operation required by the considered application domain. The CGRUs and interconnections are programmed by proper configuration (*control*) bits that are stored in configuration

memory. The configuration memory may store one or multiple configuration contexts, but at any given time, one context is active. The controller is responsible for controlling the loading of configuration contexts from the main memory to configuration memory, for monitoring the execution process of the reconfigurable hardware and for activating the reconfiguration contexts. The interconnection network can be realized by a crossbar or a mesh structure.

CGRAs can provide massive amounts of parallelism and high computational capability. Typically, the application domains of CGRAs are Digital Signal Processing (*DSP)* and multimedia. These kinds of applications usually spend most of their execution time in loop structures. These computational intensive parts have high levels of operation and data parallelism. The design of such systems requires a good correspondence between the coarse-grained reconfigurable architecture and the loop's characteristics. Kernels (*loops*) of an application are mapped onto the array in a highly parallel way. Generally, in order to schedule a kernel, it needs richer interconnections. However, richer interconnections come with costs such as wider multiplexors, more wires, and more configuration bits which translate to large silicon area and higher power consumption. Moreover, even with the same amount of interconnection resources, we can expect variation among topologies. Choosing a good topology is an essential step in the architecture exploration. Typically, the applications which belong to the application domain of the CGRAs, are characterized by the high data transfer rate between the processor and the memory [Dimitroulakos *et al.*, 2007].

0.2 Compiling Loops onto CGRAs with Modulo Scheduling

There are abundant computational resources available for parallelism in CGRAs. The target applications of CGRAs are typically telecommunications and multimedia electronics, which often spend most of their time in critical segments, typically loops [Mei *et al.*, 2003b]. The massive amounts of parallelism found in CGRAs can be used to speed up time critical loops of an application. Moreover, the loops often exhibit high degree of parallelism and require a great deal of computation intensive resources.

In order to map the critical loops, we have to consider the data dependency within an iteration of a loop and inter-iteration dependency. When compiling a loop onto CGRAs, each operation within the loop requires a resource to be executed on the CGRA and the time at which the operation will execute. The executed operation has to be routed to the dependent operations in the loop.

Since each loop iteration repeats the same pattern of executing operations, compiling loops onto CGRAs can be achieved by modulo scheduling [Hatanaka and Bagherzadeh, 2007]. Modulo scheduling is a software pipelining technique [Llosa *et al.*, 2001] that overlaps several iterations of a loop by generating a schedule for an iteration of the loop. Modulo scheduling uses the same schedule for subsequent iterations. Iterations are started at a constant interval called the Initiation Interval (II). The time taken to complete a loop of n iterations is roughly proportional to II. The main goal of modulo scheduling is to find a schedule with as low an II as possible.

The scheduling, placing and routing loops onto CGRAs faces several architectural

constraints and challenges. Modulo scheduling adds a time dimension to the combination of placement and routing, which becomes very similar to placement and routing for FPGAs [Hatanaka and Bagherzadeh, 2007].

0.3 Motivations and Objectives

In order to solve the scheduling, placing and routing problem onto CGRAs with modulo scheduling, several issues have to be considered in the mapping. A scheduling algorithm should be capable of efficiently exploiting regular data parallelism in CGRAs with lower initiation interval. The following issues motivated us to consider a modulo scheduling algorithm for CGRAs.

- An algorithm capable of achieving a lower initiation interval to start the successive iterations.
- An algorithm capable of routing intermediate data between the executed operations of loop.
- An algorithm that is fast and efficient with optimal usage of resources in the final schedule.
- An algorithm capable of mapping different execution paths of a loop caused by conditional branches.
- An algorithm able to do parallel search of solutions with placement, scheduling and routing.

- An algorithm must be able to consider the hardware constraints and conserve resources.
- An scheduling algorithm should be compatible with the front end application.
- An algorithm that is capable of mapping critical nodes and edges.
- An scheduling algorithm should be applicable to different CGRAs and different topologies.
- An algorithm that is capable of analyzing best topology of the CGRA.

Unfortunately, the available parallelism in CGRAs has been exploited by only a few automated design and compilation tools [Mei *et al.*, 2003b]. The modulo scheduling algorithm used in [Hatanaka and Bagherzadeh, 2007] and [Vassiliadis and Soudris, 2007b] was not able to find optimal usage of resources and took a long time to find the valid schedule. Several heuristic techniques were tried by researchers in solving the modulo scheduling problem, but the techniques were not fast and efficient [Llosa *et al.*, 1996]. For example, the existing scheduling algorithms find the placement and routing solution with a sequential search for each Data Flow Graph (*DFG)* operation and does not solve conditional code. Particle swarm optimization (*PSO*) applied to instruction scheduling [Abdel-Kader, 2008], provides near optimal solutions, with fast convergence and low execution time for various combinatory and multidimensional optimization problems. A simple PSO can get stuck in a locally optimal solution and can be made efficient in combination with mutation operators [Grundy and Stacey, 2008]. To the best of our knowledge, PSO has not been used in modulo scheduling

for coarse-grained architectures. As a result, a fast and efficient modulo scheduling algorithm for CGRAs with parallel search is developed.

The objectives of this thesis are:

- To develop a fast and efficient scheduling, placing and routing algorithm called modulo constrained hybrid particle swarm optimization (*MCHPSO*) to exploit loop-level parallelism of different target applications.
- To analyze the performance of MCHPSO in various CGRA topologies and configurations.
- To apply MCHPSO to various benchmarks in telecommunications and in multimedia applications and to compare the II achieved with other scheduling algorithms.
- To develop an algorithm to analyze the DFG with conditional code generated from a HARdware Parallel Objects Language (*HARPO/L*) program and to schedule the conditional code with MCHPSO with efficient use of resource.
- To develop an algorithm to handle loop-carried dependences or recurrences in DFG, where an operation depends on itself or another operation from previous iterations.

0.4 Thesis Contributions

The following are the contributions of this thesis.

- Designed the solution structure for the particles in PSO to map DFG onto a time-space graph called routing resource graph (RRG), where each particle represents a scheduling solution to the mapping process.

- Designed and implemented MCHPSO algorithm to place, schedule and route DFG onto CGRA. The algorithm succeeded in scheduling with lower initiation interval, and with minimal usage of resources. However, the MCHPSO algorithm did not conflict with any data dependency and satisfied the modulo constraints for the CGRA resources.

- Compared the performance of MCHPSO with other scheduling algorithms and analyzed MCHPSO on various topologies and various CGRA configurations, the MCHPSO algorithm achieved fast execution time and better schedule results than other algorithms. Analyzed the speedup of MCHPSO in intel i7 quad core processor. The MCHPSO parallelizes well with many logical processors and produces faster result.

- Designed and implemented a predicated exclusivity MCHPSO algorithm to map conditional code in DFG. The exclusivity algorithm was able to minimize the number of resources used in the scheduling process. The exclusivity algorithm reused the same resource for conditional code in DFG to be mapped onto CGRAs.

- Designed a preprocessing algorithm to extract information from DFG generated by the HARPO/L program compiler. The algorithm added predicates and symbolic information to the DFG cells (nodes and edges). Designed a method

to create exclusivity matrix of all DFG cells.

- Designed a method to find empty slots in MRT (modulo reservation table) using Maximum Independent Set algorithm.

- Analyzed the performance of predicated exclusive MCHPSO algorithm with various CGRA configurations. Compared the performance of predicated exclusive MCHPSO algorithm with non-exclusive predicated MCHPSO algorithm on various benchmarks.

- Implemented and evaluated a method to handle loop carried dependence in DFG to be mapped onto CGRAs.

0.5 Thesis Overview

This thesis is organized as follows. Chapter 1 provides a detailed review of modulo scheduling in CGRAs. First, an overview of CGRA has been outlined and it is followed by selecting a suitable CGRA for the selected problem. Secondly, an overview of modulo scheduling has been discussed. Thirdly, the chapter discusses evolutionary algorithms and the use of particle swarm optimization in modulo scheduling.

Chapter 2 discusses the proposed algorithm called Modulo Constrained Hybrid Particle Swarm Optimization (MCHPSO). An overview of the compilation framework has been discussed. The chapter also provides a review of the related work. The encoding of particle and fitness calculation in MCHPSO are presented in this chapter.

Chapter 3 presents the simulation results for MCHPSO. The performance analysis of MCHPSO is discussed, based on the interconnections, resource availability and

particle size. MCHPSO speedup is analyzed on the Intel i7 quad core processor.

Chapter 4 discusses the exploitation of conditional structure in CGRAs. This chapter presents the predicated exclusivity algorithm. The input DFG was taken from the HARPO/L (HARdware Parallel Objects Language) compiler and the simulation results of predicated exclusivity algorithm are discussed.

Chapter 5 presents the recurrence handling in loops. This chapter reviews various methodologies to map recurrence relations onto CGRAs. It also presents the recurrence aware prioritized MCHPSO algorithm and its simulation results.

Chapter 6 concludes the thesis and presents the scope for future work.

Chapter 1

Compilation in Coarse-Grained Reconfigurable Architectures

1.0 Introduction

Coarse-grained reconfigurable architectures (*CGRAs*) have the potential to exploit both the efficiency of hardware and flexibility of software to map large applications. A good compiler should employ the CGRA's resources to exploit a high amount of operation and loop-level parallelism in the application's loops [Tuhin, 2007]. The compiler must carefully schedule the application's loop body and facilitate high performance at a reasonable cost.

An overview of CGRAs and the selection of target architecture is given in Section 1.1. Compiling loops to CGRAs involves the modulo scheduling process which is a combination of 3 tasks: scheduling, placement, and routing which will be discussed in Section 1.2. In this thesis, the modulo scheduling is done with particle swarm op-

timization. The various kinds of evolutionary algorithms and the reason for selection of PSO are discussed in Section 1.3. This chapter concludes with a discussion of the different compilation procedures attempted so far in the CGRAs and the need for a new modulo scheduling algorithm in Section 1.4.

1.1 Coarse-Grained Reconfigurable Architecture

1.1.0 Introduction

Coarse-Grained Reconfigurable Architectures have been used widely for accelerating time consuming loops. Processing elements (*PEs*), available in a large number of CGRAs, can be used to exploit the inherent parallelism found in loops to accelerate the execution of applications. In a CGRA, the PEs are organized in a 2-dimensional (2D) array, connected with a configurable interconnect network [Dimitroulakos *et al.*, 2009].

1.1.1 Overview of some CGRAs

1.1.1.0 MorphoSys

The MorphoSys architecture has been designed for multimedia applications to accommodate applications with data parallelism and high throughput constraints, such as video compression [Singh *et al.*, 2000a]. The components of the MorphoSys architecture are an array of reconfigurable cells (*RCs*), processing units (*called RC Array*), a general-purpose (*core*) processor (*TinyRISC*) and a high-bandwidth memory interface, implemented as a single chip. The computation-intensive operations are handled

by the single instruction multiple data (*SIMD*) array of coarse-grained reconfigurable cells (*CGRCs*). The sequential processing and the RC array operation controls are performed by the TinyRISC [Singh *et al.*, 2000b]. A context word is loaded into the RC's context register for every execution cycle.

1.1.1.1 KressArray

KressArray (*also known as rDPA*) has a 32-bit-wide data path with an array of reconfigurable processing elements. The KressArray reconfigurable architecture features arithmetic and logic operators on the level of the C programming language, making the mapping simpler than for FPGAs [Hartenstein *et al.*, 2000]. It consists of a mesh of PEs, also called rDPUs (*reconfigurable Data Path Units*), which are connected to each of their 4 nearest neighbors by 2 bidirectional links with a data path width of 32–bits, where "bidirectional" means a direction is selected at configuration time.

1.1.1.2 Montium

The coarse-grained reconfigurable part of the Chameleon system-on-chip is called the Montium Tile [Heysters and Smit, 2003]. The Montium Tile is especially designed for mobile computing and targets the 16-bit digital signal processing (*DSP*) algorithm domain [Smit *et al.*, 2007]. Montium supports both integer and fixed-point arithmetic, with a 16-bit datapath width. The tile is interfaced with the outside world with the communication and configuration unit (*CCU*). The tile has 5 identical arithmetic and logic units (ALU1...ALU5) that can exploit spatial concurrency to enhance performance. Dedicated input output units (*DIOs*) are used to handle fast and parallel transfers of input/output data that are placed around the array

architecture [Alsolaim *et al.*, 1999].

1.1.1.3 DReAM

Dynamically reconfigurable architecture for mobile systems (*DReAM*) [Alsolaim, 2002] was designed to be a part of a system-on-a-chip (*SoC*) solution for the third and future generations of wireless mobile terminals. It consists of an array of concurrently operating coarse-grained reconfigurable processing units (*RPUs*). Each RPU was designed to execute all required arithmetic data manipulations and control-flow operations. To perform fast dynamic reconfiguration, the configuration memory unit (*CMU*) holds configuration data for each of the RPUs and is controlled by one responsible communication switching unit (*CSU*).

1.1.1.4 CHESS

The reconfigurable arithmetic array (*RAA*), termed CHESS [Marshall *et al.*, 1999], was developed by hewlett packard (*HP)* Labs to provide high computational density, wide internal data bandwidth, distributed registers, and memory resources for important multimedia algorithm cores. CHESS also offers strong scalability, software flexibility and advanced features for dynamic reconfiguration. CHESS's functional units are 4-bit ALUs and it reduces the number of bits of configuration memory by having 4-bit bus connections. It allows a small configuration memory to speedup reconfiguration.

1.1.1.5 RaPiD

RaPiD [Ebeling, 2002] is a coarse-grained reconfigurable architecture to achieve the low cost and high power efficiency of application-specific integrated circuits (*ASICs*), without losing the flexibility of programmable processors. RaPiD architecture is configured to form a linear computational pipeline, with a linear array of functional units (*FUs*). Each RaPiD cell contains 3 ALUs, one multiplier, three 32-word local memories, 6 general-purpose "datapath registers" and 3 small local memories. The RaPiD array is designed to be clocked at 100MHz and reconfiguration time for the array is conservatively estimated to be 2000 cycles [Ebeling *et al.*, 1997].

1.1.1.6 PipeRench

PipeRench [Goldstein *et al.*, 2000] is a reconfigurable fabric with a network of interconnected configurable logic and storage elements. PipeRench contains a set of physical pipeline stages called stripes. In each stripe, the interconnection network accepts inputs from each processing element in that stripe and one of the register values from each register file in the previous stripe. Each PE contains an arithmetic logic unit (*ALU*) and a pass register file where the ALU contains lookup tables (*LUTs*) and extra circuitry for carry chains, zero detection, and so on. PipeRench was designed to improve reconfiguration time, compilation time, and forward compatibility, increased flexibility, reduced chip development and maintenance fabrication costs.

1.1.1.7 ADRES

The architecture for dynamically reconfigurable embedded systems (*ADRES*) [Mei *et al.*, 2005a] tightly couples a very long instruction word (*VLIW*) processor and a reconfigurable array. The architecture has 2 virtual functional views: the VLIW processor view and the reconfigurable array view built into a single architecture [Mei *et al.*, 2003b]. The VLIW processor, consisting of several functional units and a multiport register file (*RF*), serves the first row of the reconfigurable array. Some FUs in the first row can connect with memory to facilitate data access for load/store operations. The reconfigurable array is intended to efficiently execute only computationally intensive kernels of applications [Mei *et al.*, 2003a]. The architecture template, shown in Figure 1.0, consists of many basic components, including computational, storage, and routing resources.

The FUs can execute a set of word-level operations selected by a control signal. Register files and memory blocks can store intermediate data. Routing resources, including wires, multiplexers, and buses connect the computational resources and storage resources defined by the topology through point-to-point connections or a shared bus. The different instances of the architecture can be generated by a script-based technique and by specifying different values for the communication topology, the supported operation set, resource allocation, and latency in the target architecture [Zalamea *et al.*, 2004].

The results can be written to the distributed RFs, which are small and have fewer ports than the shared RF, or they can be routed to other FUs. An output register buffers each of the FU's outputs, to guarantee timing. Multiplexers are used to route

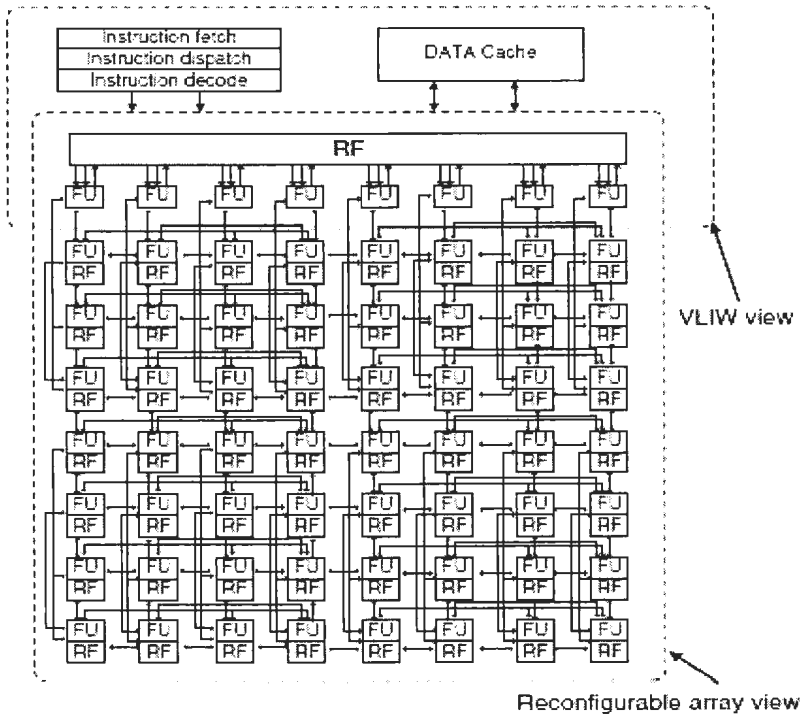


Figure 1.0: ADRES Architecture taken from [Mei *et al.*, 2005c]

data from different sources. The configuration RAM stores the configuration for each cycle. In ADRES, the integration of predicate support, distributed register files and configuration RAM make it applicable and efficient to many applications.

1.1.2 Comparison and Selection of the Target CGRA

The various CGRAs discussed above have their advantages and disadvantages. MorphoSys has a 16-bit granularity with mesh based structure, fast memory interface, dynamic programming and requires a manual placement and routing tool [University of California, 2009]. KressArray has a highly flexible mapper used to map massively

communication-intensive applications [Hartenstein *et al.*, 2000] and provides area efficient and throughput efficient design. KressArray can be used only for limited applications with regular data dependencies [Becker *et al.*, 1998]. Montium focuses on providing sufficient flexibility, provides abundant parallelism, but has limited configuration spaces [Guo, 2006]. ADRES uses the VLIW processor for non-kernel code and reduces the communication cost between the VLIW and reconfigurable matrix through the shared RFs for resource sharing [Vassiliadis and Soudris, 2007a]. DReAM was designed for modern wireless communication system and provides an acceptable trade off between flexibility and application performance [Becker *et al.*, 2000].

CHESS offers strong scalability, dynamic reconfiguration but it has a constraint that the ALU and switchbox should be of the same size and the need of long wires for the transfer of data [Marshall *et al.*, 1999]. RaPiD features static and dynamic control to map a range of applications but it has the disadvantage of a data path with an implicit directionality [Ebeling, 2002]. PipeRench trades off configuration size for compilation speed by hardware virtualization and improved compilation time, reconfiguration time, and forward compatibility. PipeRench has a low bandwidth between main memory and processor, which limits the type of applications which require speed up [Goldstein *et al.*, 2000].

Among the various coarse-grained architectures discussed, the ADRES architecture was considered for the proposed research. The reason for this choice was that the ADRES architecture is a flexible architecture template, with low communication costs. The loops present in an application can be mapped onto the ADRES array in a highly parallel way with ease of programming. The compiler within the ADRES template is automatically retargetable i.e., it has been designed to be relatively easy

to modify to generate code for different configurations and have provided a good deal of data for comparison.

1.2 Scheduling

1.2.0 Introduction

The objective of scheduling is to minimize the execution time of a parallel computation application by properly allocating tasks to the processors by avoiding the processor stall cycles. Scheduling inner loop bodies is a NP-hard problem which implies that there is no polynomial time algorithm that can give an optimal solution to the problem (assuming $P \neq NP$) [Kwok and Ahmad, 1999]. The ultimate goal of scheduling is to create an optimal schedule, a schedule with the shortest length of the given application. Schedule length or makespan is measured as the overall execution-time of a parallel program in cycles. Additionally, when a schedule is produced, the scheduling algorithms must satisfy both resource and precedence constraints.

Depending on the constraints, scheduling may be broadly classified into 3 main categories [$ChingandKeshab, 1995$].

Time-Constrained Scheduling minimizes the number of the required resources when the iteration period is fixed.

Unconstrained Scheduling does not have any fixed timing or resource usage during the scheduling.

Resource-Constrained Scheduling fixes the number of resources and the objectives to determine the fastest schedule, or the smallest iteration period.

List scheduling is the most commonly used scheduling approach. It can be classified under resource constrained scheduling and time constrained scheduling. A scheduling list is statically constructed before node allocation begins, and most importantly, the sequencing in the list is not modified. List scheduling is often used for both instruction scheduling and processor scheduling [Beaty, 1994]. In an iteration, nodes with a higher priority are scheduled first and lower priority nodes are deferred to a later clock cycle based on the priority functions like as soon as possible ($ASAP$), as late as possible ($ALAP$), mobility, height-based priority etc. [Tuhin, 2007]. The priority sorting is carried out by selecting a node based on the priorities listed above and added to the priority sort list. The sorting is then carried out for each child node of the selected node until all the nodes in the list are processed.

1.2.1 Software Pipelining

Software pipelining [Lam, 1988] is a scheduling technique which overlaps the operations in the successive iteration to yield processors's fast execution rate. Software pipelining is a global cyclic scheduling problem to exploit the instruction level parallelism (ILP) available in loops. The idea is to look for a pattern of operations from various iterations (*often termed as the kernel*) so that when repeatedly iterating over this pattern, it produces the effect that iterations are initiated at a regular interval. This interval is termed the initiation interval (II). Thus successive iterations of the loop are in execution with different stages of their computation. Once a schedule is obtained, the loop is reconstructed into a prologue, a kernel, and an epilogue. Instructions in the prologue are repeated until the pipeline is filled. The prologue consists of

code from the first few iterations of the loop. The loop kernel or steady state [Allan *et al.*, 1995] consists of instructions from multiple iterations of the original loop, and a new iteration of the kernel is initiated at every II cycles. Instructions in the epilogue are designed to complete the functionality of code and consist of code to complete the last few iterations of the loop.

1.2.2 Modulo Scheduling

Modulo scheduling [Mei *et al.*, 2003a] is a software pipeline technique which overlaps several iterations of a loop by starting successive iterations at a regular interval. The main goal of modulo scheduling is to simplify the process of software pipelining by generating a schedule for an iteration of the loop and use the same schedule for subsequent iterations at constant intervals. Modulo Scheduling ensures that it satisfies data dependence constraints and intra- and inter-iteration dependency, and no resource availability conflicts.

The schedule for an iteration is divided into stages so that different stages of the successive iteration execution get overlapped. The number of stages in an iteration is called its stage count (SC), and the number of cycles per stage is termed the initiation interval. The Initiation Interval should be minimized to exploit as much parallelism from a loop as is possible and modulo scheduling tries to minimize it [Tuhin, 2007].

The II is constrained either by loop-carried dependences of the loop (i.e cases where data from an earlier iteration is used in a later iteration) or by resource constraints of the hardware. The limit on the II set by loop-carried dependence is called recurrence minimal initiation interval ($RecMII$), while the limit set by resource constraints is

called resource minimal initiation interval (*ResMII).* The minimal initiation interval (*MII*) is a lower bound to start the pipeline scheduling process and it is computed as $MII = \max(ResMII, RecMII)$ [Llosa *et al.*, 2001]. If a valid schedule cannot be obtained by an II equal to MII, then II is incremented by one and the scheduling process is repeated until a valid schedule is obtained or the algorithm gives up.

Modulo scheduling can be illustrated by taking an example of the dependence graph shown in Figure 1.1b, along with a 2×2 architecture. The data dependence graph unrolled for 3 iterations, is shown in Figure 1.1a. The initiation interval is 1 and so at time cycle 2, all the 3 iterations are executing at different stages.

A modulo schedule can be generated by the use of heuristics and integer linear programming. Since modulo scheduling is based on heuristics, it may not always give the optimal solution. There are many heuristic algorithms developed for modulo scheduling such as

- Iterative modulo scheduling [Rau, 1994]
- Recurrence cycle aware modulo scheduling [Oh *et al.*, 2009]
- Clustered modulo scheduling [Sánchez and González, 2001]
- Swing modulo scheduling [Llosa *et al.*, 1996]
- Hypernode reduction modulo scheduling [Llosa *et al.*, 1995]
- Modulo scheduling with integrated register spilling [Zalamea *et al.*, 2001].

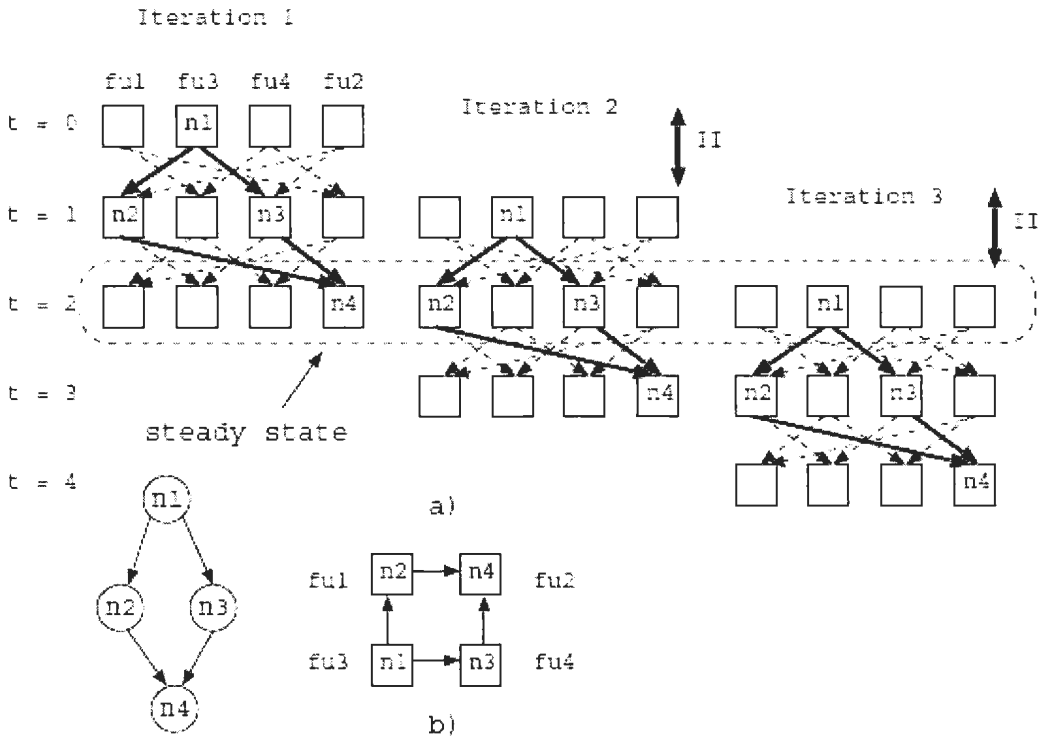


Figure 1.1: a)Modulo Scheduling Example b)DFG and Configuration for 2x2 matrix, modified from [Mei *et al.*, 2003b]

1.2.3 Graph Embedding

Graph embedding is a problem in graph theory [Newsome and Song, 2003] in which a directed guest graph $G_1 = (V_1, E_1)$ is embedded in another directed host graph $G_2 = (V_2, E_2)$ [Heath, 1997]. The embedding consists of a one to one function ρ_v from $V1$ to $V2$ and a function ρ_e that maps each edge (u, v) ε E_1 to a path in G_2 between $\rho(u)$ and $\rho(v)$. There are 3 kinds of primary cost, measured in graph embedding: dilation, expansion, and congestion [Heath, 1997]. For a given embedding (ρ_v,ρ_e), the congestion of edge e_2 in G_2 is the number of edges e_1 in G_1 such that e_2 is on the

path $\rho_e(e_1)$; the congestion of an embedding is its maximum edge congestion. The length of the longest assigned path is called the dilation of the graph embedding. The ratio $\frac{|V_2|}{|V_1|}$ is called the expansion of the graph [Heath, 1997]. Using graph embedding, the performance of one network (guest graph) over another network (host graph) can be investigated. Graph embedding provides a systematic approach to various node-node communication problems [Newsome and Song, 2003]. The concept of graph embedding can be extended to solve many problems [Guattery and Guattery, 1997]. Graph models are successfully employed in various applications such as computer aided circuit layout, network topologies, data-centric applications in sensor networks, and so on [Newsome and Song, 2003], [Levi and Luccio, 1971]. Graph embedding is effective in scheduling, placing and routing because it can take into account the communication structure of the loop body and scales well with respect to the number of operations [Park *et al.*, 2006].

1.2.4 Modulo Reservation Table

In software pipelining, the modulo reservation table (MRT) is used in determining if there is a resource conflict while allocating resources. MRT can be used to represent the resource usage of the steady state by mapping the resource usage at time t to that at time t mod s [Lam, 1988].

1.2.5 Routing Resource Graph

When modulo scheduling is applied to the data flow graph, the intermediate operands are routed by allocating resources in the routing resource graph (RRG) [Ebeling *et*

al., 1995]. The RRG is replicated from the architecture graph for every time cycle. RRG reserves resources by enforcing modulo constraints.

1.3 Evolutionary Algorithms

In order to find a scheduling, placing, and routing for the loops in CGRAs, we have to find a valid schedule with the minimum number of resource usage and with the smallest possible II and also satisfy all dependence and modulo constraints. Some approaches have been tried to schedule loops, such as with simulated annealing [Mei *et al.*, 2005c],[Hatanaka and Bagherzadeh, 2007] to minimize the number of resources used in routing. In this section, some selected evolutionary algorithms will be discussed briefly and we will conclude with the selection of an evolutionary algorithm for our modulo scheduling algorithm.

1.3.0 Overview

1.3.0.0 Simulated Annealing

Simulated annealing (*SA*) [Wang *et al.*, 2001] is a method to solve global optimization problems, with a metaheuristic approach, to the global minimum of a given function in a large search space. The term simulated annealing, is derived from the roughly analogous process of heating and controlled cooling of a material to increase the size of its crystals and reduce the number of defects to obtain a strong crystalline structure [Fang, 2000]. SA is often used when the search space of the problem is continuous. SA can accept worse neighboring solutions, with a certain probability that depends on a variable called the temperature (T). In the SA method, the temperature T is

gradually reduced as the simulation proceeds. Initially, T is set to a high value (or infinity), and it is decreased based on a reduction ratio r, which is close to 1, at each time step and ends with $T = 0$ at the end of the allotted time budget. The simulated annealing process is stopped when the system reaches a frozen solution state, that is when there is no improvement in the solution configurations.

1.3.0.1 Genetic Algorithm

Genetic algorithms (GAs) were originally developed by John Holland and his research students. GA is the most widely used evolutionary computation technique [Uysal and Bulkan, 2008]. GA operates on strings of data in which each string represents a solution, in a way that resembles a chromosome in natural selection. Genetic algorithm exhibits implicit parallelism because they analyze and modify a set of solutions simultaneously [Song *et al.*, 2008].

Genetic algorithms generate random solutions as the initial population. There are 3 stochastic operators applied to the population.

Selection Is a portion of the existing population selected to breed a new generation of population.

Crossover Is a genetic operator that generates new offsprings by randomly choosing some crossover point and everything before this point is copied from a first parent and then everything after a crossover point copy from the second parent., which hopefully retain good features from the parents.

Mutation Is a genetic operator that randomly modifies the new offspring with a probability. It can enhance the diversity of the population and provide a chance

to escape from local optima.

In a long run of a GA, the better (lower cost) solutions tend to stay in the population and the worse (higher cost) solutions tend to disappear [Uysal and Bulkan, 2008] in accordance with the theory of survival of the fittest. Genetic algorithms are able to solve large problems with parallel nature. GA has been applied to various fields such as neural networks, data mining, electronic circuit design, scheduling applications and so on [Davis, 2010].

1.3.0.2 Ant Colony Optimization

Ant colony optimization (*ACO*) [Dréo *et al.*, 2006], which takes inspiration from the foraging behavior of some ant species, has been formalized into a metaheuristic for combinatorial optimization problems. The original ant colony optimization algorithm was known as ant system (AS) [Dorigo *et al.*, 1996] and was proposed in the early nineties. Each ant in the AS is a possible solution to the problem. Certain ants lay down an initial trail of pheromones to mark some favorable path as they return to the nest with food. A pheromone is a chemical signal that triggers a natural response to attract other ants and serves as a guide. In the meantime, some ants do random exploratory survey for closer food sources. Ant Systems make a probabilistic decision by implementing a randomized construction heuristic. ACO has inherent parallelism and gives positive feedback for good solutions. ACO can be applied to telecommunication networks, graph coloring, scheduling, constraint handling and so on [Shekhawat *et al.*, 2009].

1.3.0.3 Particle Swarm Optimization Algorithm

Particle swarm optimization (*PSO*) [Kennedy and Eberhart, 1995] is an optimization approach that follows an evolutionary metaphor. It is a population-based search procedure in which individuals, called particles, change their positions, or states, with time. Particles in a PSO system move in a multidimensional search space [Abdel-Kader, 2008] to find a good solution. All the particles can share their information about the search space with other particles. During the process, each particle modifies its position in the search space according to its best experience and the experience of nearby particles, and makes use of the best position met by it and other neighboring particles [Chen and Sheu, 2009].

A detailed explanation of PSO is given in this section, as this algorithm will be used in the proposed modulo scheduling algorithm. The reason for choosing PSO is explained in the next subsection. Each particle in a PSO system represents a potential solution to the problem. At the end of the search, the best particle will hold the best solution found. At every iteration, each particle k calculates its velocity and position according to the expressions given below.

$$V_{k,i+1} = w \times V_{k,i} + c_1 r_1 (P_{k,i} - X_{k,i}) + c_2 r_2 (P_{g,i} - X_{k,i}) \tag{1.0}$$

$$X_{k,i+1} = X_{k,i} + V_{k,i+1} \tag{1.1}$$

where

- $X_{k,i}$ - Particle k coordinates at i^{th} iteration

- $X_{k,i+1}$ - Particle k coordinates at $i + 1^{th}$ iteration
- $V_{k,i}$ - Velocity of particle k at i^{th} iteration
- $V_{k,i+1}$ - Velocity of particle k at $i + 1^{th}$ iteration
- c_1, c_2 - acceleration constants in range [0, 1]
- r_1, r_2 - random value in range [0, 1]
- $P_{k,i}$ - Particle k's personal best position found at i^{th} iteration
- $P_{g,i}$- Global best particle position at i^{th} iteration
- w - Inertia weight factor. It is calculated by

$$w = w_{\max} - \frac{w_{\max} - w_{\min}}{i_{\max}} \times i \tag{1.2}$$

where

- $w_{\min}$ and $w_{\max}$ are both random numbers called minimum weight and maximum weight respectively.
- $i_{\max}$ is the maximum number of iterations

After calculating X_{i+1}, we can get the new particle position to search in the next iteration. Each particle velocity on each dimension is limited to the maximum velocity.

In most cases, all the particles tend to converge to the best solution quickly. PSO has a strong search capability in the problem space and can save more computation

$i := 0, k := 0$
For each particle in the PSO system
    Initialize particle with random coordinates.
    Initialize current particles $X_{k,i}$ coordinates as
    the particles best position $P_{k,i}$
End
$i := 0$
Do
For each particle in the PSO system
    Calculate fitness value of the given particle.
    If the fitness value is better than the best
    fitness value pbest in history set current
    coordinates value as the new $P_{k,i}$
End
Choose the particle with the best fitness value of all the
particles as the P_g
For each particle
    Calculate particle velocity using Equation (1.0)
    Update particle position using Equation (1.1)
End
$i := i + 1$
While maximum iterations is not attained.

Algorithm 1.0: The Standard PSO Algorithm

time for finding an acceptable solution [Wang *et al.*, 2007]. The selection of parameters $c1$ and $c2$ affects the performance [Chen and Sheu, 2009].

1.3.1 Selection of PSO Algorithm

When PSO was used in the Traveling Salesperson Problem (TSP), PSO showed a significant performance in the initial iterations when compared with ACO [Nonsiri and Supratid, 2008]. PSO has the capability to quickly arrive at an optimal or a near-optimal solution. ACO has a difficult theoretical analysis, sequence of random decisions, and uncertain convergence time [Shekhawat *et al.*, 2009].

An advantage of PSO over GA is that PSO maintains all the solutions in the search space and changes of inertia weight leads to convergence [Nonsiri and Supratid, 2008]. PSO keeps the history of all the particles local best fitness and the global best fitness. When a particle gets caught in a bad solution, it can still go to its previous best position and start searching. GA is stochastic and contains no information about the problem [Thenorio, 2010]. GA can prematurely converge to a local optimum solution in its reproduction process rather than the global optimum of the problem [Thenorio, 2010]. This suggest trying PSO on modulo scheduling.

The relative ease of implementation and the ability to provide reasonably good solutions are the advantages of simulated annealing, but it takes a great deal of computation time and a careful tuning of parameters [Elmohamed *et al.*, 1998] to obtain good solutions. The PSO method has the advantages of fast speed to get the solutions, stable convergence and robustness and it is a parallel direct search method to generate good solutions [Song *et al.*, 2008]. PSO can be applied to various fields,

for example, to train artificial neural networks, function optimization, fuzzy control system and so on [Hu, 2009]. PSO parameters are so designed that they are highly adaptive [Acharjee and Goswami, 2009].

Previous research on PSO [Abdel-Kader, 2008],[T.Chiang *et al.*, 2006] shows that instruction scheduling can be done with PSO, in this thesis, PSO with a hybrid combination of mutation operation is tried. The mutation operator is used in the proposed modulo scheduling algorithm to avoid premature convergence in PSO algorithm.

1.4 Various CGRA Compilation Procedures

In recent years, the compilation of applications, written in a high-level language to coarse-grained reconfigurable platforms, has become the subject of research. Computationally intensive kernels present in the application are represented by data flow graphs (DFGs) where nodes represent the operations and edges form the communication between the nodes. Some selected compilation procedures are discussed in detail in the subsequent sections. The compilation procedure differs with different mapping algorithms, target architecture representations and handling constraints during compilation. Compiling applications to CGRAs involve 3 tasks: Scheduling, Placement, and Routing. Scheduling assigns the time cycle to execute the operation. Placement assigns a functional unit and Routing takes care of moving data from producer functional unit to consumer functional unit.

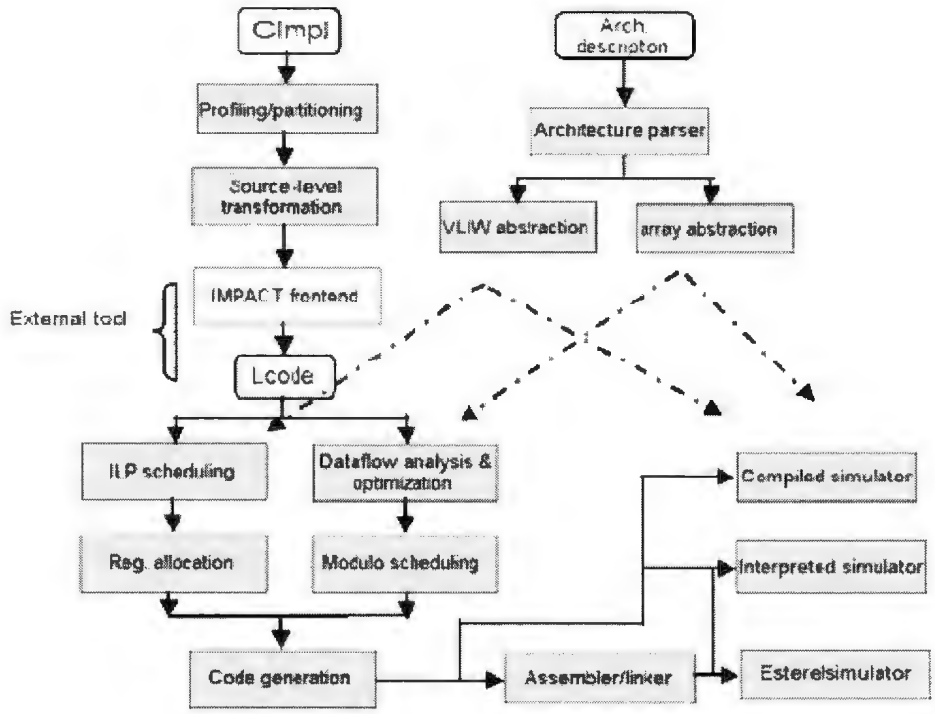


Figure 1.2: DRESC Compiler Framework, taken from [Berekovic *et al.*, 2006]

```
SortOps();
II   := MII(DDG);

while not scheduled do
  InitMrrg(II);
  InitTemperature();
  InitPlaceAndRoute();

  while not scheduled do
    for each op in sorted operation list
      RipUpOp();

      for i := 1 to random_pos_to_try do
        pos := GenRandomPos();
        success := PlaceAndRouteOp(pos);

        if success then
          new_cost := ComputeCost(op);
          accepted := EvaluateNewPos();
          if accepted then
            break;
          else
            continue;
          endif
        endfor

      if not accepted then
        RestoreOp();
      else
        CommitOp();

      if get a valid schedule then
        return scheduled;
    endfor

    if StopCriteria() then
      break;

    UpdateOverusePenalty();
    UpdateTemperature();

  endwhile
  II++;
endwhile
```

Figure 1.3: Pseudocode of the modulo scheduling algorithm in DRESC, taken from [Mei *et al.*, 2002]

1.4.0 DRESC Compiler

Dynamically reconfigurable embedded system compiler (*DRESC*) [Mei *et al.*, 2002], is a retargetable compiler that is able to parse, analyze, transform, and schedule plain C source code to the DRESC [Mei *et al.*, 2005a] architecture. Figure 1.2 presents the overall structure of the DRESC compiler. Source-level transformations are done on the target C source code to rewrite the kernel in order to make it pipelinable and to maximize the performance of the functional units. The target architecture is described in an extensible markup language (*XML*). The parser and abstraction steps transform the architecture into an internal graph representation called a modulo routing resource graph (*MRRG*), which is used by the modulo scheduling algorithm.

Modulo scheduling plays a central role in the DRESC compiler, by creating high parallelism for the kernels [Vassiliadis and Soudris, 2007a]. The task of modulo scheduling is to produce a software pipeline schedule with a low initiation interval. A MRRG [Mei *et al.*, 2003b] is introduced in DRESC to model the architecture internally for the modulo scheduling algorithm. The MRRG combines features of the modulo reservation table [Lam, 1988] and the routing resource graph [Ebeling *et al.*, 1995]. The MRRG is a directed graph showing the time space representation of the architecture which is constructed by composing sub-graphs representing the different resources of the ADRES architecture [Mei *et al.*, 2003a]. The pseudocode of the modulo scheduling algorithm is given in Figure 1.3. The modulo scheduling algorithm generates an initial schedule respecting dependency constraints for each II. The inner loop of the algorithm uses simulated annealing to minimize the overuse of resources. If simulated annealing succeeds within the time allotted then the loop is

exited. If the algorithm cannot find a valid schedule in the time budget, it tries with an incremented II.

1.4.0.0 Advantages and Limitations

The modulo scheduling algorithm of the DRESC compiler has some limitations. For large loop bodies, it has long convergence time due to the use of simulated annealing. It does not scale well with the size of the DFGs because while taking scheduling decisions, it does not take any information from the DFG. The overall performance is adversely affected when there is a spare interconnection among the FUs. Moreover, the scheduling algorithm only considers the innermost loop of a nested loop structure [Tuhin, 2007].

1.4.1 Compilation using Modulo Graph Embedding

A graph theoretic technique called modulo graph embedding (*MGE*) is used in [Park *et al.*, 2006] for compiling applications to CGRAs. Using MGE, loop bodies are mapped onto CGRAs, subject to modulo resource usage constraints. The communication structure of the loop body was considered during mapping to make it an effective technique. Initially, preprocessing was done to analyze the DFG and to construct a skewed scheduling space. A skewed scheduling space does not allow all the FU slots to be available at the given schedule time. The start times of FUs are restricted such that they stagger down the right side of the CGRA. The skewed scheduling space dynamically changes as operations gets placed in an FU. The scheduling space of all the FUs to the right of the placed FU are lowered to guarantee the routability.

The next step in the mapping process is followed by the main scheduling loop

to find a placement for all the operations at a particular height of the DFG using modulo graph embedding. The scheduling process first constructs the affinity graph for the given input DFG. Next the primary slots are identified to place, schedule and route. The scheduler enters an inner loop to determine the cost of the current layout and iteratively reduces the cost using simulated annealing.

1.4.1.0 Advantages and Limitations

Modulo Graph Embedding [Park *et al.*, 2006] uses a skewed scheduling space and a systematic placement decision with a search space limited to the same height. The method achieves good convergence and fast compilation times. This technique is not suitable for DFG with loop-carried dependencies, as these dependencies are given the same priority as intra-iteration dependencies [Oh *et al.*, 2009].

1.4.2 Compilation using Clustering

Montium architecture [Guo, 2006] presents a framework for scheduling clusters written in a high-level language (*C++*). In this work, the scheduling problem is called the color-constrained scheduling problem where the limitations of using processors resources with one-ALU and 5-ALU configurations are termed as color and pattern. The color-constrained scheduling problem was tackled in this work by 3 algorithms: the multi-pattern scheduling algorithm, the column arrangement algorithm and the pattern selection algorithm. The multi-pattern scheduling algorithm, used in this work, is similar to the list scheduling algorithm, with extra constraints. The algorithm schedules the node in the colored graph G. The successors of a node should be scheduled after the node has been scheduled. The column arrangement algorithm

orders the non-ordered pattern elements. In the pattern selection algorithm, a non-ordered pattern color bag, is selected.

1.4.2.0 Advantages and Limitations

Scheduling clusters with Montium architecture exploits the high speed parallelism of the source code and consumes low energy with few clock cycles. The performance of the algorithm has to be improved to refine the priority functions and to decrease the computation complexity due to a large number of candidates [Guo, 2006]. The number of iterations of a loop in the DFG was not clearly specified and they did not consider loop carried dependences of a loop.

1.4.3 Compilation Using Modulo Scheduling with Backtracking Capability

The compilation approach described in [Dimitroulakos *et al.*, 2009] presents an exploration framework that automates the evaluation of CGRA architectures. The framework, used in this approach identifies the CGRA architectures tuned for a specific application domain with modulo scheduling. In this work, an effective priority scheme is proposed while the modulo scheduler has been equipped with backtracking capability. The loop schedule is constructed by mapping incrementally one operation at a time in the loop. There are 4 steps which comprise the scheduling loop: priority scheme, enforce dependence constraints, enforce resource constraints, and mapping decision selection.

1.4.3.0 Advantages and Limitations

The mapping algorithm suggested in this work, proposes to reduce congestion and map all the operations to PEs and effectively route the data values between PEs. The experiments carried out through this algorithm, indicate that the algorithm has a significant impact on the performance. The architecture's performance affects the clock frequency and instructions per cycle (*IPC*). A higher IPC value not only has a negative impact on the clock frequency but it also requires more hardware [Dimitroulakos *et al.*, 2009].

1.5 Conclusion

This chapter discussed the background literature of the proposed research. The various CGRAs, designed during the past years, have been discussed. The existing architecture ADRES is considered for the proposed work because of its flexible topology and simple implementation. The basic scheduling techniques were discussed. The basic modulo scheduling technique will be used in the proposed work. Among the various evolutionary algorithms discussed, PSO will be used in the proposed work for the following reasons: it is easy to implement, it has fast convergence, and it is efficient.

Finally, some of the compilation approaches published, were discussed. The compilation frameworks for the CGRAs have some commonality such as constructing an acceptable application graph, abstracting information from the target architecture and mapping to make the best use of the resources available in a CGRA. A compilation framework similar to the work done in [Mei *et al.*, 2003a] is taken for the proposed

research. The modulo scheduling is carried out by the Particle Swarm Optimization, with a mutation operator.

Chapter 2

Modulo Constrained Hybrid Particle Swarm Optimization Scheduling Algorithm

2.0 Introduction

This chapter gives an overview of the MCHPSO scheduling algorithm. The research focuses on modulo scheduling algorithms for CGRAs. The study of various compilation frameworks in CGRAs, as discussed in Chapter 1, indicates that not much work has been done to improve the basic modulo scheduling algorithm to map onto CGRAs more effectively. We need to efficiently use the reconfigurable resources available in CGRAs and to keep the time low to schedule complex target applications. To solve the modulo scheduling problem in CGRAs, an algorithm is proposed in Section 2.2. The MCHPSO algorithm schedules, places, and routes an inner loop body represented

by a data flow graph (DFG). The DFG is embedded into the routing resource graph (RRG) of the target CGRAs by using particle swarm optimization (PSO), combined with a mutation operator. The background concepts related to the MCHPSO algorithm are discussed in Section 2.1. The different steps of the MCHPSO algorithm are discussed in Section 2.2. The evaluation, applications and comparisons of the MCHPSO algorithm are discussed in Section 2.3.

2.1 Modulo Scheduling in CGRAs

2.1.0 Problem Identification

The objective of modulo scheduling is to find a valid schedule of one iteration of the loop body so that it may be repeated at regular intervals. The schedule must respect all intra-iteration and inter-iteration dependency and resource constraints and economically use the resources and execution time [Mei *et al.*, 2003a]. The number of clock cycles between the start of successive iterations is termed the schedules's initiation interval (II), essentially reflecting the performance of the scheduled loop. The problem of determining the lowest possible initiation interval, and a schedule that meets it, for a given loop on a given hardware is an NP-hard problem and therefore there is no known way to efficiently solve it in all cases.

Several heuristic techniques have been tried to solve the modulo scheduling problem. A heuristic technique solves problems based on experience and randomization, and uses repeated random sampling to compute the results. Nature-inspired, a bird-flocking experience-based technique is used in the heuristic approach of the PSO algo-

rithm for problem solving and discovery, which may be applied to problems which are time-consuming to find a solution. When PSO is compared with ant colony optimization (ACO) [Dorigo *et al.*, 2006], PSO shows the ability to quickly arrive at an optimal or near-optimal solution. An advantage of PSO over genetic algorithms (GA) [Uysal and Bulkan, 2008] is that PSO maintains all the solutions in the search space and requires less computational effort to arrive at high quality solutions. Previous research [Abdel-Kader, 2008], [Xiaoyu Song and Cao, 2008] on PSO shows that scheduling can be done with PSO. We enhanced PSO with a hybrid combination of mutation operator for our modulo scheduling problem to avoid premature convergence in PSO algorithm. Our early results showed that using PSO to solve the scheduling problem gives a near-optimal solution. When PSO is combined with a randomization method, discovering the near optimal solution becomes faster and the local optimal solution can be avoided. This combination of heuristic approach and randonization is what we term modulo constrained hybrid PSO (MCHPSO). This is a practical approach to solve the scheduling problem. The proposed algorithm is discussed in detail in the following subsections.

2.1.1 Solution Structure Formalization

Most applications in mobile computing and multimedia spend a lot of time repeatedly executing critical code segments called loops. Since iterations of these loops can often be executed in parallel, we can make effective use of the abundant resources available in CGRAs. After mapping a loop onto the CGRA, we can use the free resources in the CGRA for another application or loop kernels. To increase the free resources of

the CGRAs, we need a mapping algorithm that produces a valid schedule with a low routing cost.

To address the problem of mapping a loop body of a target application onto CGRAs, we propose a modulo scheduling algorithm by using a PSO algorithm combined with a random mutation operator. The schedule length of the loop is its total execution time in cycles. If a resource r of total resources (R) in the routing resource graph (RRG) (described in Section 2.1.1.4) at time t (in clock cycle) is being used, it is reserved for all times t' such that $t \equiv t' \,(\mathrm{mod}\ II)$(. The unavailability of the same resources for successive iterations is called a modulo constraint. While scheduling loops, the algorithm has to satisfy the dependence constraints on each operator involved in the loop and not violate modulo constraints to start the successive iterations in parallel. To illustrate the overall problem, an example is shown in Figure 2.0.

The following conditions should be satisfied while scheduling a loop:

0. In a loop body, if an operation y depends on the result of operation x, then the operation x is not scheduled to end later than operation y starts.

1. Operations which are independent of each other may be executed in parallel.

2. When a resource is occupied by an operation, it is reserved for all equivalent times ($mod\ II$) of the schedule length.

3. If a computational resource is free, it can also be used as a routing resource.

In Figure 2.0, the inner most loop of the target application is converted into a DFG by using static single assignment (SSA) and dependence analysis (*explained in next subsection*). The target architecture (TA) is created as a graph by using the

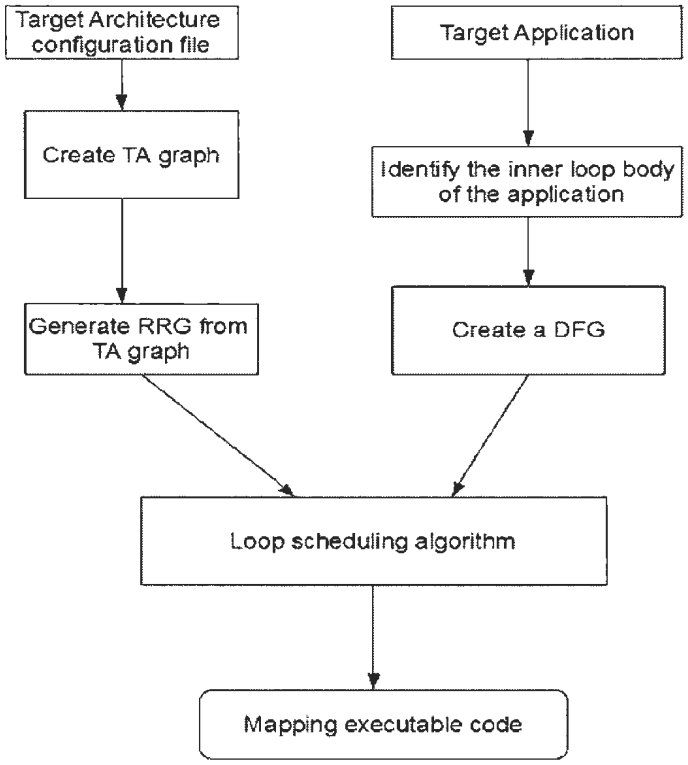


Figure 2.0: Outline of overall mapping of loop kernel of DFG onto RRG of CGRA

topology and resource constraints specified in a description file. The TA is replicated to the maximum possible schedule length to form the RRG. The RRG contains edges between replications to represent data carried forward in time. Now the mapping algorithm tries to map each node of the DFG to a node of the RRG and each edge of DFG to a path in the RRG. An iteration of the target application is placed, routed and scheduled by satisfying modulo constraints to repeat the same schedule at every initiation interval (II) for the consecutive iterations.

2.1.1.0 Data Flow Graph

The target application program is analyzed to find the critical loops to be mapped onto the CGRA. In this chapter, the inner loop body of the application is considered with no inter-iteration dependence and no nested loop dependence to explain with a simple DFG. In the later chapters, we consider mapping in the presence of conditions and recurrences in the DFG. The inner loop body of the application is called its loop kernel. From the loop kernel, we created a data flow graph representation of $DFG = (N, E, \overleftarrow{}, \overrightarrow{})$ [Tuhin and Norvell, 2008] where

- N : Set of operations in the inner loop body.
- E : Set of interconnection edges.
- $\overleftarrow{}$: is a function mapping each edge e to its source node $\overleftarrow{e}$.
- $\overrightarrow{}$: is a function mapping each edge e to its target node $\overrightarrow{e}$.

The loop kernel is analyzed by converting it into static single assignment (*SSA*) form and then using dependence analysis to convert it into a DFG. In the SSA form, every variable is assigned exactly once. The dependence analysis explains the dependence between operations, showing which operation should be executed first i.e, Predecessor (*Pred*) and which operation is the successor (*Succ*). Each edge of the DFG has 2 parameters (*delay,distance*) which are shown in Figure 2.1. The delay (λ) is the processing time of the source node and the distance is the difference in the iteration number between source and target nodes. If both the source and target nodes are in the same iteration, the distance is denoted by 0. The DFG in Figure

2.1 shows the recurrence in the loop forming a circuit (C) from *node Z* to Z through node *c1* and *Op X*.

The II calculation is discussed in the next subsection. As the complexity of the DFG increases, the total number of nodes and total number of edges to be mapped, also increases. For each operation of the loop, we created a predecessor list and successor list. When the nodes and edges are created in the DFG, a longest path delay priority sort is applied on the DFG to create an edge list. During the routing process in the mapping algorithm, the edges are routed according to the list order. To schedule the operations of DFG, the ASAP (as soon as possible) time, ALAP (as late as possible) time and mobility are calculated for each operation u in the DFG according to the following equations.

$$ASAP_u = \begin{Bmatrix} ASAP_u = 0; \text{ if Pred(u)} = \emptyset \\ ASAP_u = \max(ASAP_v + \lambda_v) \ ; \forall v \in \text{Pred(u)} \end{Bmatrix} \tag{2.0}$$

$$ALAP_u = \begin{Bmatrix} ALAP_u = \max(ASAP_v) \ ; \ v \in V \\ ALAP_u = \min(ALAP_v - \lambda_v) \ ; \forall v \in \text{Succ(u)} \end{Bmatrix} \tag{2.1}$$

$$Mobility\_u = ALAP_u - ASAP_u \tag{2.2}$$

2.1.1.1 Target Architecture

In this thesis, a wide range of CGRAs are targeted, as discussed in Chapter 1. The target description file contains enough information about the various resources, constraints on each resource and their interconnections of the TA graph. To start with, an architecture similar to the ADRES [Vassiliadis and Soudris, 2007b] architecture

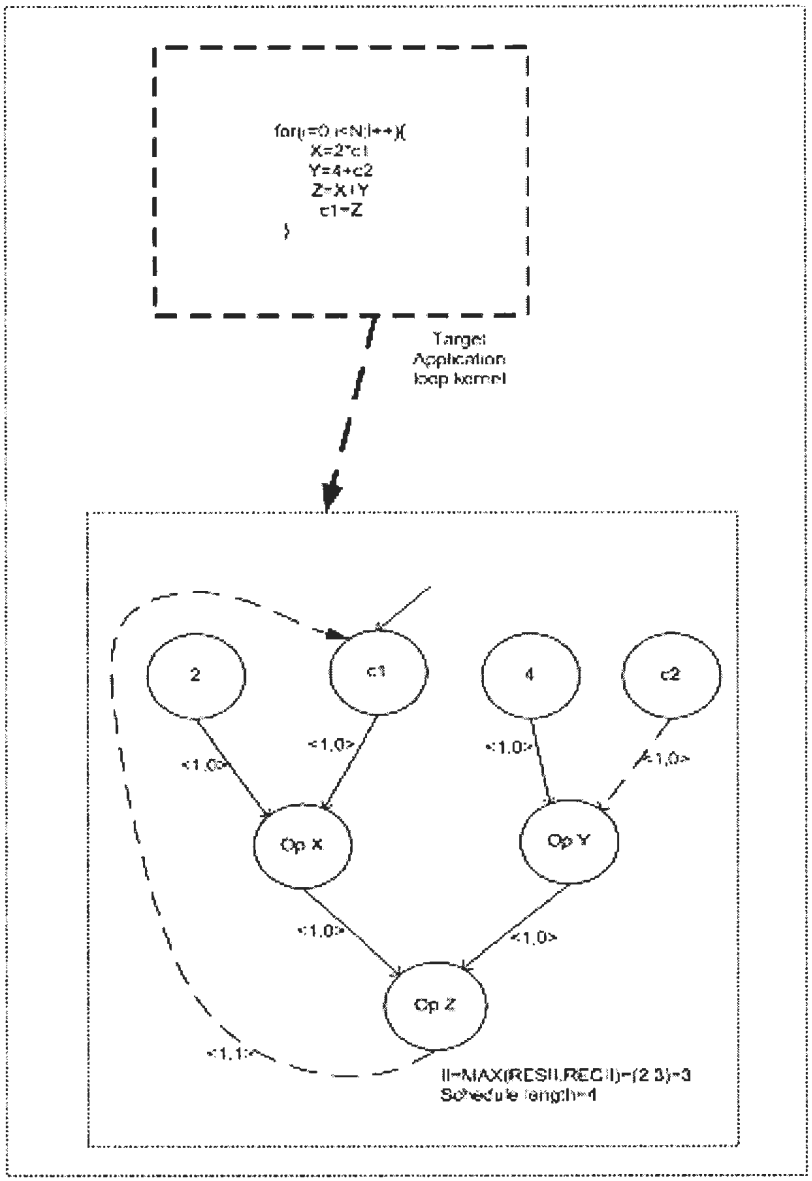


Figure 2.1: A loop body converted into a DFG

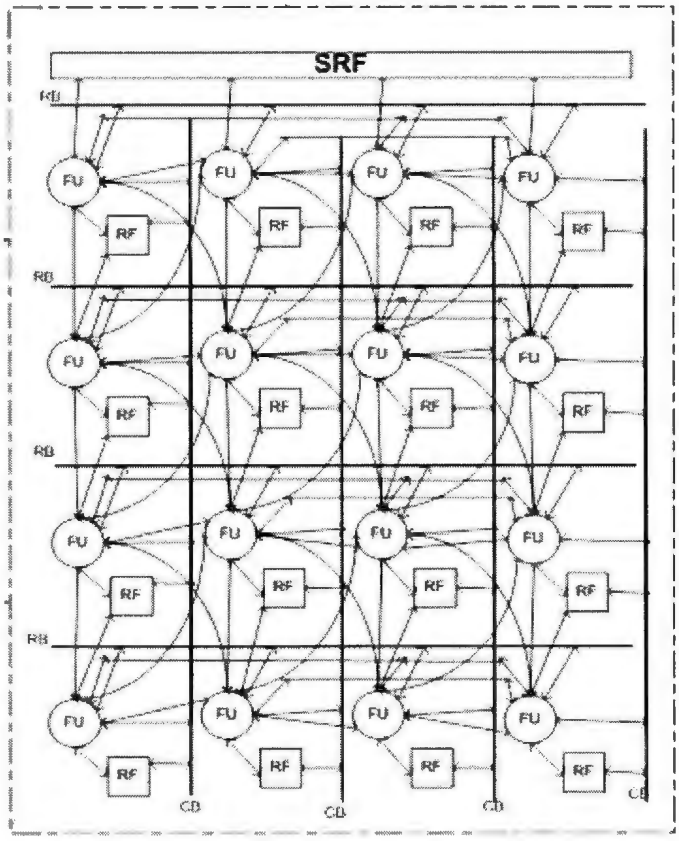


Figure 2.2: 4×4 Target Architecture Instance of ADRES.

template is targeted. The reason for this choice was that the ADRES is a flexible architecture template with low communication costs. The loops present in an application can be mapped onto the ADRES array in a highly parallel way. The compiler within the ADRES template is automatically retargetable, i.e., it has been designed to be relatively easy to modify to generate code for different architectures. The target architecture forms a 2D array and it is a very flexible template specified in the description file. The TA consists of a network of basic components, including functional units (*FUs*), register files (*RFs*), column buses (*CBs*), and row buses (*RBs*).

The TA graph (V, S) is formed from the target description file where:

V is a set of vertices. Each vertex can represent any of the resources mentioned above. Each vertex is described by its name, capacity, and its functionality.

S is a set of directed edges, connecting pairs of resources in the graph.

Each FU can receive input from various resources of the graph and similarly the output of each FU can be routed to various destination resources [Vassiliadis and Soudris, 2007b]. The various topologies for the FUs are displayed in Figure 2.3: There are (a) a mesh based architecture of 4 neighboring FU connections; (b) a meshplus1 architecture of 8 neighboring FU connections; and (c) a meshplus2 architecture of 4 neighboring FU connections along with row connection for every FU and column connection for every FU.

Various topologies of TA, including register files, are presented in Figure 2.4, which (a) shows each FU having its own private RF; (b) shows each RF is shared by the FUs in the top and bottom row of the same column; (c) shows each FU has a RF and the RF is shared among FUs adjacent in all the diagonal directions.

Various uses of buses are exhibited in Figure 2.5: (a) shows usage of row buses where each FU is connected to its corresponding row bus; (b) shows usage of both row and column buses where each FU is connected to its corresponding row bus and each FU and RF is connected to its corresponding column bus.

The computational resources are FUs, which are capable of executing a set of coarse-grained operations such as add, subtract, multiply, and shift and can also forward information, when not performing any operation. The top row of FUs can only perform load and store operations, termed as memory unit (MU). The storage

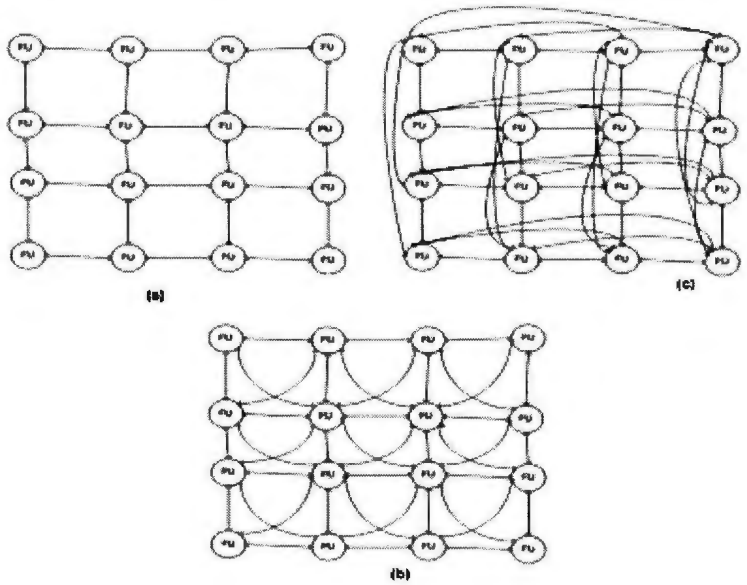


Figure 2.3: FU Topology (a) Mesh Topology (b) Meshplus1 Topology (c) Meshplus2 Topology

resources mainly refer to the RFs which can store intermediate data and multi-ported shared RFs (*SRF*) used by the MUs to hold the load and store values from the memory [Mei *et al.*, 2003b]. The routing resources include wires, CBs and RBs to connect various computational and storage resources.

The results of each FU can be routed to other FUs through direct connections or routing resources or may be written to an RF for routing at a later time. If the FUs are free without executing any operations, then they may be used for routing purposes. The number of registers in the RF can be specified in the description file. In our RF, each register has 2 read ports and 1 write port.

The target architecture taken for the MCHPSO algorithm is shown in Figure 2.2 and includes meshplus2 FU connections shown in Figure 2.3c, shared RF shown in

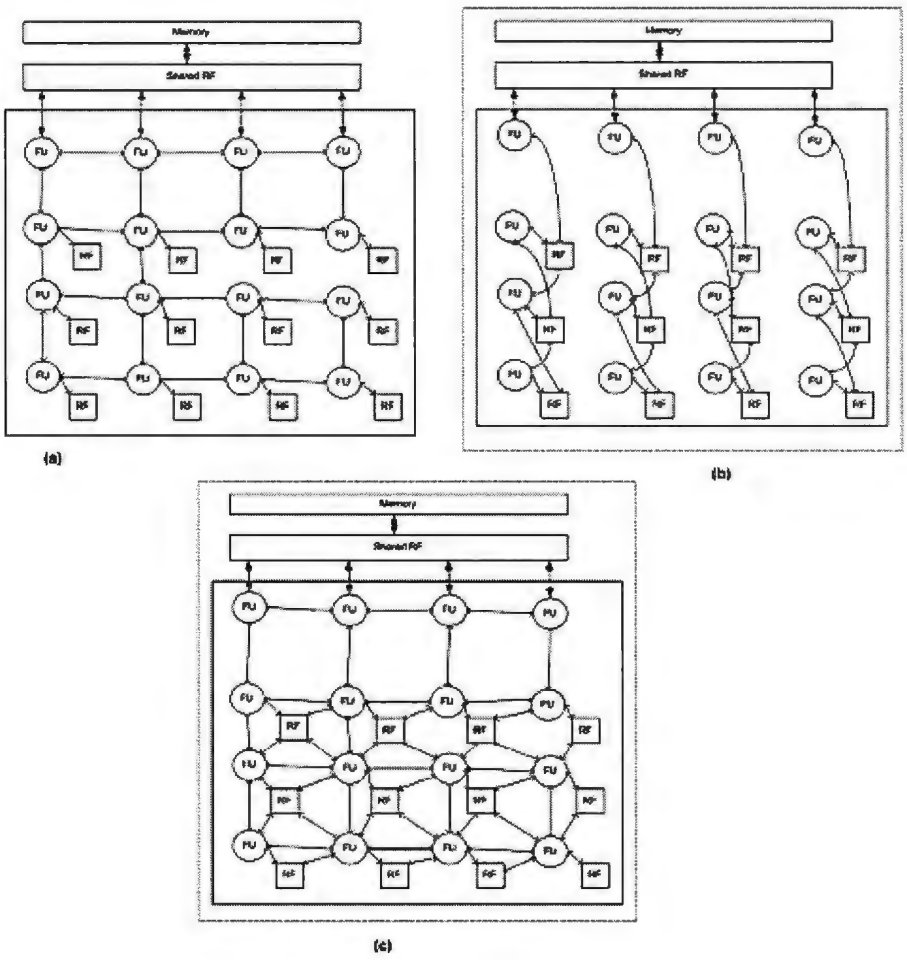


Figure 2.4: FU and RF Topology (a) Private RF (b) Private RF and Column Adjacent Topology (c) Private RF and Diagonal Adjacent Topology.

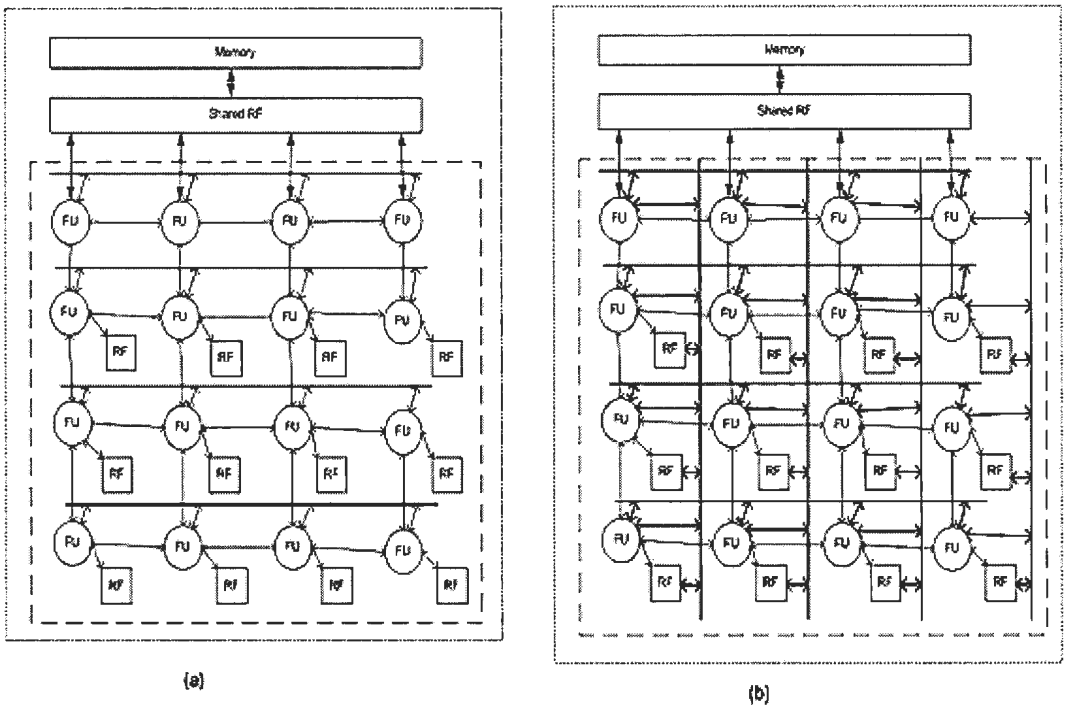


Figure 2.5: Various Usage of Buses (a) Row Bus Connections (b) Row and Column Bus Connections

Figure 2.4c, and usage of both row and column buses shown in Figure 2.5b.

2.1.1.2 Minimal Initiation Interval

As discussed in Chapter 1, the minimal initiation interval (MII) is the larger value from ResMII and RecMII, as computed in

$$MII = \max(ResMII, RecMII) \tag{2.3}$$

where,

- Resource minimal initiation interval (ResMII) is calculated from the resource

usage requirements of the loop and it is derived from

$ResMII = \text{Max}_{r \in R} \left\lceil \frac{\#r\ needed}{\#r\ available} \right\rceil$, where r is a category of resource in the TA resources R.

- Recurrence minimal initiation interval (RecMII) is calculated from the recurrence cycles in the DFG. Each recurrence have a distance property, which is equal to the number of iterations separating the 2 instructions involved in the recurrence. If a dependence edge, $e(v, u)$, in a cycle has latency λ and connects the operations at $\delta_{v,u}$, then the RecMII is calculated by RecMII = $\text{Max}_{c \in C} \left\lceil \frac{l}{d} \right\rceil$ where,

  - c is a recurrence cycle in the set of all cycles C of the DFG
  - l is the sum of all delay (λ) in the circuit
  - d is the sum of all distance $\delta_{v,u}$ in the circuit, variable $\delta_{v,u}$,denotes the distance between operation v and u, which means the operation u of iteration i depends on the operation v of iteration $i - \delta_{v,u}$.

  In our algorithm, the availability of MU resources is checked for each load or store memory operation in the DFG. An example of the MII calculation is shown in Figure 2.1.

2.1.1.3 Modulo Reservation Table

To enforce the modulo constraints, we have to generate a schedule for one iteration of the loop in such a way that the same schedule is repeated at regular intervals with respect to data dependence and resource constraints [Vassiliadis and Soudris, 2007b].

This interval is called the initiation interval, as defined in the previous subsection. The II reflects the performance of the scheduled loop. The modulo reservation table (MRT) is constructed as a table, with one column per each resource in the TA and II rows shown in Table 2.0 for the DFG and TA shown in Figure 2.1. For every new placement (schedule and place) and update in RRG, the MRT is checked to determine whether the time/place is available. If the mapped node in the RRG uses (v, t_i), then $(v, t_i \bmod II)$ in the MRT is filled, which marks the resource v busy for all times with the same modulus by $\{(v, t_j) \mid t_j \bmod II = t_i \bmod II\}$ where $j \in \{0, SchLength\}$ [Vassiliadis and Soudris, 2007b].

Table 2.0: MRT showing all the resources occupied in II time

Resources/II time	F1	F2	F3	F4	RF1	RF2	RB1	RB2	CB1	CB2
0	Op x	Op y	x-z	y-z						
1			Op z		x-z	y-z				

2.1.1.4 Resource Routing Graph

For scheduling the loops in the DFG in a 2D architecture array across time, we employed a time-space graph called routing resource graph (RRG). Based on [Vassiliadis and Soudris, 2007b] and [Tuhin and Norvell, 2008], we produced a graph to route resources between the scheduled and placed operations across time. The RRG is obtained from the TA graph described above by replicating it once for every time cycle $\in \mathbb{N}$, specifying the interconnections with X, Y, and Z edges. The RRG is described below

$RRG = (V \times \mathbb{N}, X \cup Y \cup Z)$

where

$V \times \mathbb{N}$: An infinite set of $\mathbb{N}$ copies of the TA's vertex set V .

X **edges:** Every incoming edge e of a FU or RB or CB in the TA graph from the FU or RF is replicated across time as $X = \{x(t,e) \mid e \in E, \overleftarrow{e} \in FU \cup RB \cup CB, \overrightarrow{e} \in FU \cup RF, t \in \mathbb{N}\}$ where $\overleftarrow{x(r,e)} = (\overleftarrow{e}, t)$ *and* $\overrightarrow{x(r,e)} = (\overrightarrow{e}, t)$. Here x is simply some one-one function to a set of edges, i.e. a function that generates a unique edge for each time and TA edge.

Y **edges:** Every incoming edge e of a RF in the TA graph from the FU or CB or RB is represented in the RRG as an outgoing edge from its source in the current time cycle to the RF, CB and RB in the next time cycle. Use of such an edge represents the writing to a register or the delay in latch to the buses [Vassiliadis and Soudris, 2007b]. These RRG edges are given by edge $Y = \{y(t,e) \mid e \in E, \overleftarrow{e} \in RF, \overrightarrow{e} \in FU \cup CB \cup RB, t \in \mathbb{N}\}$ where $\overleftarrow{y(r,e)} = (\overleftarrow{e}, t)$ and $\overrightarrow{y(r,e)} = (\overrightarrow{e}, t+1)$. Here y is some one-one function to a set of edges, which is range-disjoint from x.

Z **edges:** For every RF r in the TA graph, we needed to hold the data across time. Hence we need RRG edges $Z = \{z(t,r) \mid r \in RF, t \in \mathbb{N}\}$ where $\overleftarrow{z(r,t)} = (\overleftarrow{r}, t)$ and $\overrightarrow{z(r,t)} = (\overrightarrow{r}, t+1)$, and z is a one-one function, range disjoint from x and y.

In the actual implementation, we can get away with representing only a finite prefix of the RRG, as the number of nodes in the DFG are finite and known. In order

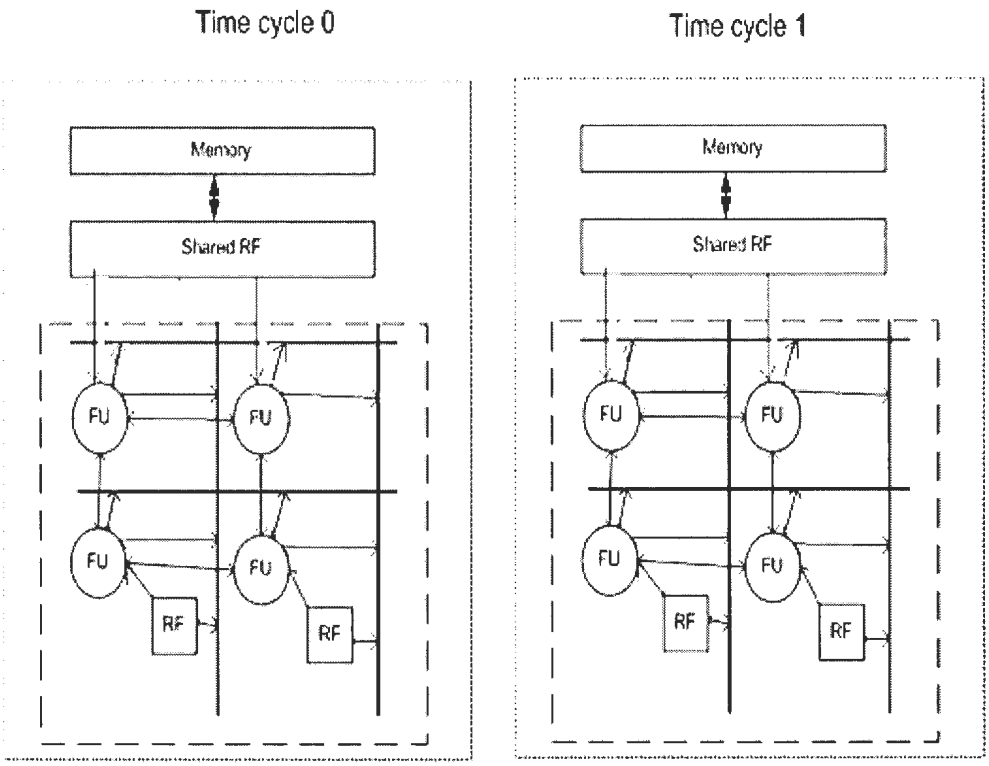


Figure 2.6: X edges in the RRG

to schedule, place, and route, we must embed the DFG into the RRG. A homeomorphism h maps nodes of the DFG to nodes of the RRG and edges of the DFG to paths in the RRG. If $h(e) = e_0, e_1, ..., e_{k-1}$, then we required $\overleftarrow{e_0} = h(\overleftarrow{e})$ and $\overrightarrow{e_{k-1}} = h(\overrightarrow{e})$, where h is the mapping of nodes and the mapping of nodes in RRG satisfies one to one and onto properties. There are further constraints to ensure that resources are not overused and that different iterations of the same loop do not interfere when they execute in parallel. An RRG corresponding to the TA in Figure 2.2 are illustrated by the X edges in Figure 2.6, the Y edges are shown in Figure 2.7, and the Z edges are shown in Figure 2.8.

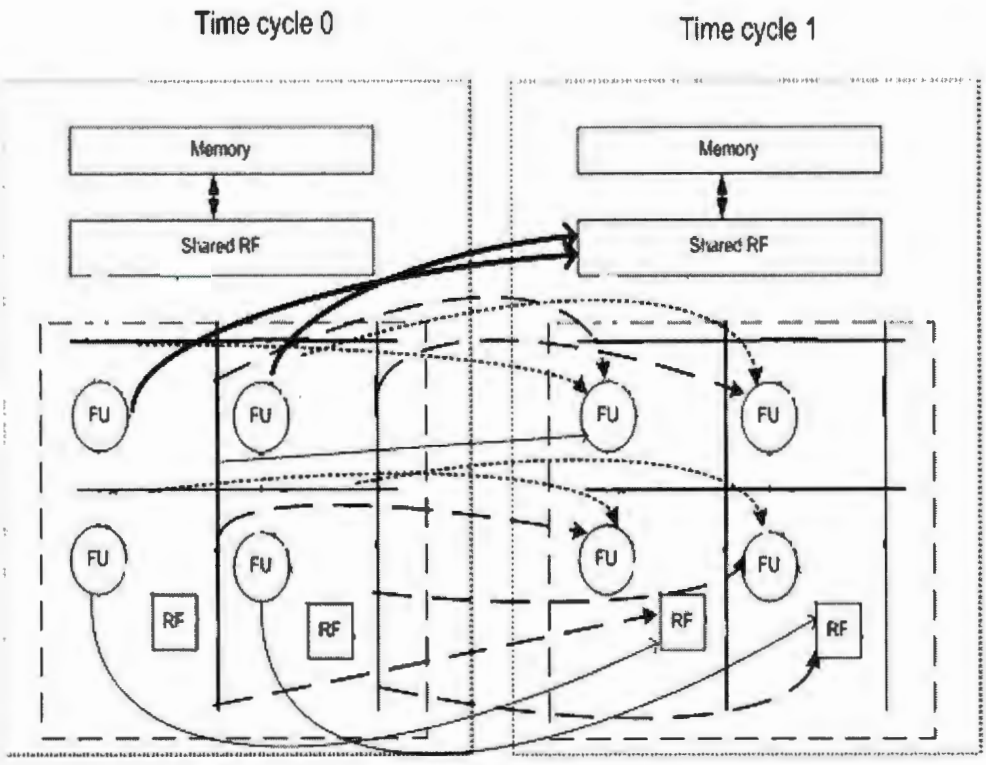


Figure 2.7: Y edges in the RRG. Edges from same type of source are shown in same style edge.

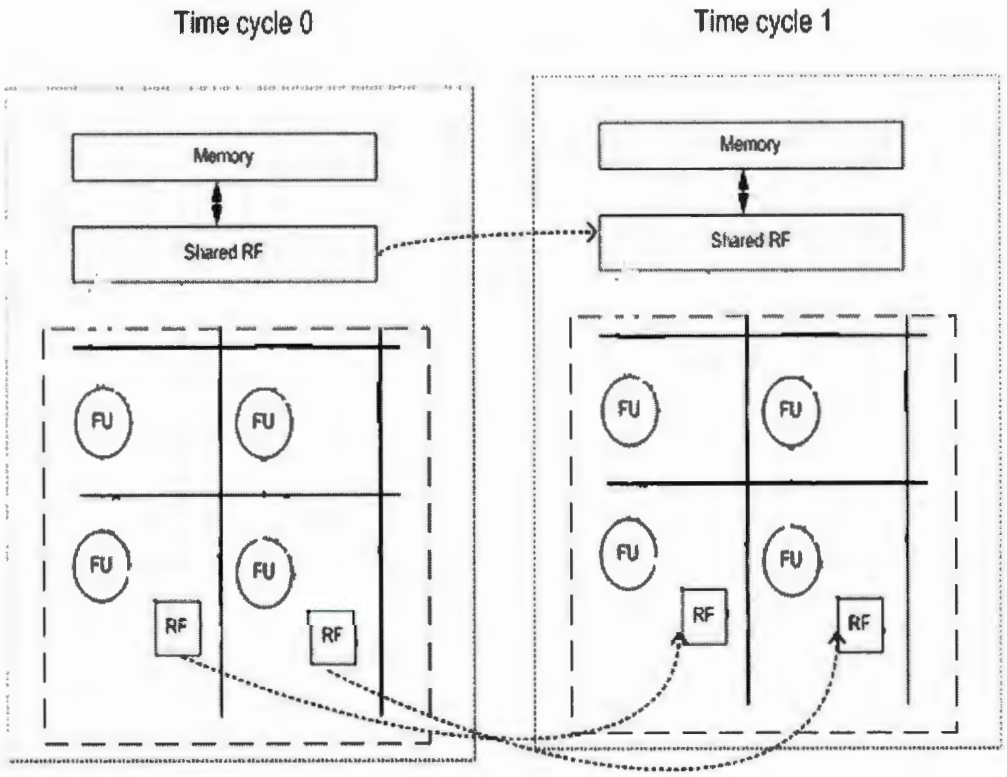


Figure 2.8: Z edges in the RRG

2.2 Proposed Modulo Scheduling Algorithm

2.2.0 Modulo Scheduling with MCHPSO

The proposed MCHPSO scheduling algorithm simultaneously searches for a good schedule, placement, and routing solution for the entire set of operations given in a DFG and it also avoids the time consuming sequential search for each operation as done in list scheduling [Mei *et al.*, 2003a]. Earlier work by Mei et al [Mei *et al.*, 2003a], Tuhin and Norvell [Tuhin and Norvell, 2008], and Vassiliadis and Soudris [Vassiliadis and Soudris, 2007b], needed several trials to find the best schedule for an operation before proceeding to the next operation. In the proposed algorithm, all the particles search for a complete scheduling solution simultaneously.

To efficiently map loops onto the CGRA, the idea of modulo scheduling used in [Mei *et al.*, 2003a] has been adopted and combined with 2 heuristic approaches, PSO and randomization. From [Abdel-Kader, 2008] and [T.Chiang *et al.*, 2006], it is noted that PSO could be applied to multidimensional scheduling problems. The application of PSO to modulo scheduling converges faster, but can be caught in a local optimum [Uysal and Bulkan, 2008]. To escape local optima, a randomization method, in combination with PSO was employed.

In MCHPSO, the routing of particles is done by using Dijkstra's algorithm [9]. To ensure modulo constraints and a valid schedule, the fitness function is computed to evaluate the quality of placement. While calculating the fitness function, routing cost of all paths routed between placements was incorporated.

The overall method of MCHPSO to schedule, place and route a loop is explained in Algorithm 2.0. The inputs to Algorithm 2.0 are a TA graph of the architecture

template and a DFG representing the inner loop part of an application. The results of the algorithm are the scheduled time, resource placement, and routing paths of an iteration of the loop.

First, the minimum initiation interval is computed as discussed in subsection 2.1.1.2. Second, ASAP (*as soon as possible*) and ALAP (*as late as possible*) times were calculated as in Equations (2.0 and 2.1) [Llosa *et al.*, 2001] for the given DFG to create a *dfglist*. Next the edges to be routed were sorted using *sort* method according to the critical path delay of the loop and the maximum schedule length is calculated from the maximum ALAP with a relaxation factor using *findschLength* method. The relaxation factor is the time cycle adjustment to place and route the leaf nodes. The relaxation factor can vary for different DFGs during the experiment setup. The RRG is generated from the TA graph. The initial placement, schedule and route may overuse resources. The MCHPSO algorithm is used to reduce overuse with a minimal routing cost. Now, starting with the minimal initiation interval the MCHPSO is used to try to find a good scheduling, placement and routing at successively larger initiation intervals. The flow of the MCHPSO algorithm is described in Figure 2.12.

2.2.1 Particle Encoding for the Problem

To frame the solution for the scheduling problem by using the particles, various dimensions for each particle, size of DFG and the schedule time should be considered. To establish a complete modulo scheduling solution, the particles of PSO were created with multiple dimensions to solve the critical issues in specified problem domain. It is necessary to search for a good-quality candidate solution for the scheduling problem,

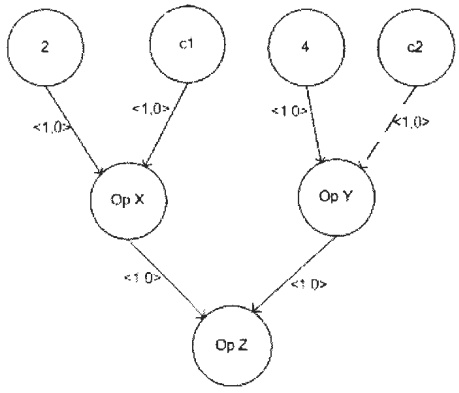


Figure 2.9: DFG showing a simple loop structure without recurrence

and then to choose the best candidate solution into the next iteration according to various objectives mentioned in the fitness function.

Therefore, the particles are encoded as an array of vectors, where each vector represents a particle. In the swarm, each particle P is represented by a mapping from the N nodes of the DFG to a RRG nodes, i.e., to time/resource pairs, as explained in Figure 2.13, and an array list to hold the routing path of each of the edges in the DFG.

2.2.2 MCHPSO

The pseudocode is shown in Algorithm 2.1. In MCHPSO, inputs are the RRG, the sorted DFG from the main loop of the ModuloSch_Place_Route Algorithm 2.0, and a goal II.

The number of operations in the DFG is initialized to the number of nodes, N, for each particle. Each particle in the PSO takes, for each node initial value for the place and a randomly chosen initial time in the range of $[ASAP,\ ALAP]$ that satisfies all

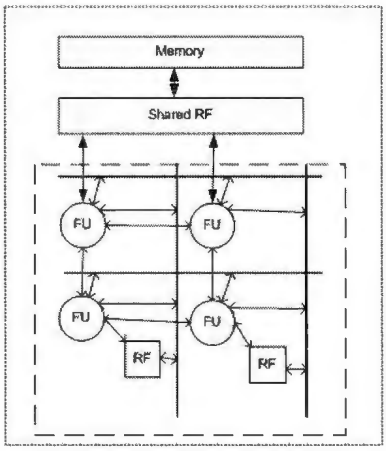


Figure 2.10: TA taken for the mapping of DFG

dependence constraints. When a resource at time t is occupied in MRT, it is reserved for every cycle with the same modulus (*with respect to II*) in the RRG. Once all the particles are initialized, the following is repeated a fixed number (*NLOOPS*) of times: First the fitness of each particle is calculated, as illustrated in the next subsection. Next every particle updates its Local-best (*Lbest*) position if the new fitness is better than the current fitness and it is denoted by P_{Lbest}. Once all the particles have been updated, the global particle of the most fit schedule is chosen and its position is denoted by P_{Gbest}.

Every particle updates to its new position according to the following

$$\begin{aligned} &\text{if } \textit{flip1} \text{ then } P_{new} = P_{Gbest} \\ &\text{else if } \textit{flip2} \text{ then } P_{new} = P_{Lbest} \\ &\text{else } P_{new} = currentP_i \end{aligned} \tag{2.4}$$

$$\text{DFG\_Rop} = \text{Random\_op}(P_{new}) \tag{2.5}$$

$$\text{avail\_slots} = \text{MRT\_check}(DFG\_Rop, P_{new}) \tag{2.6}$$

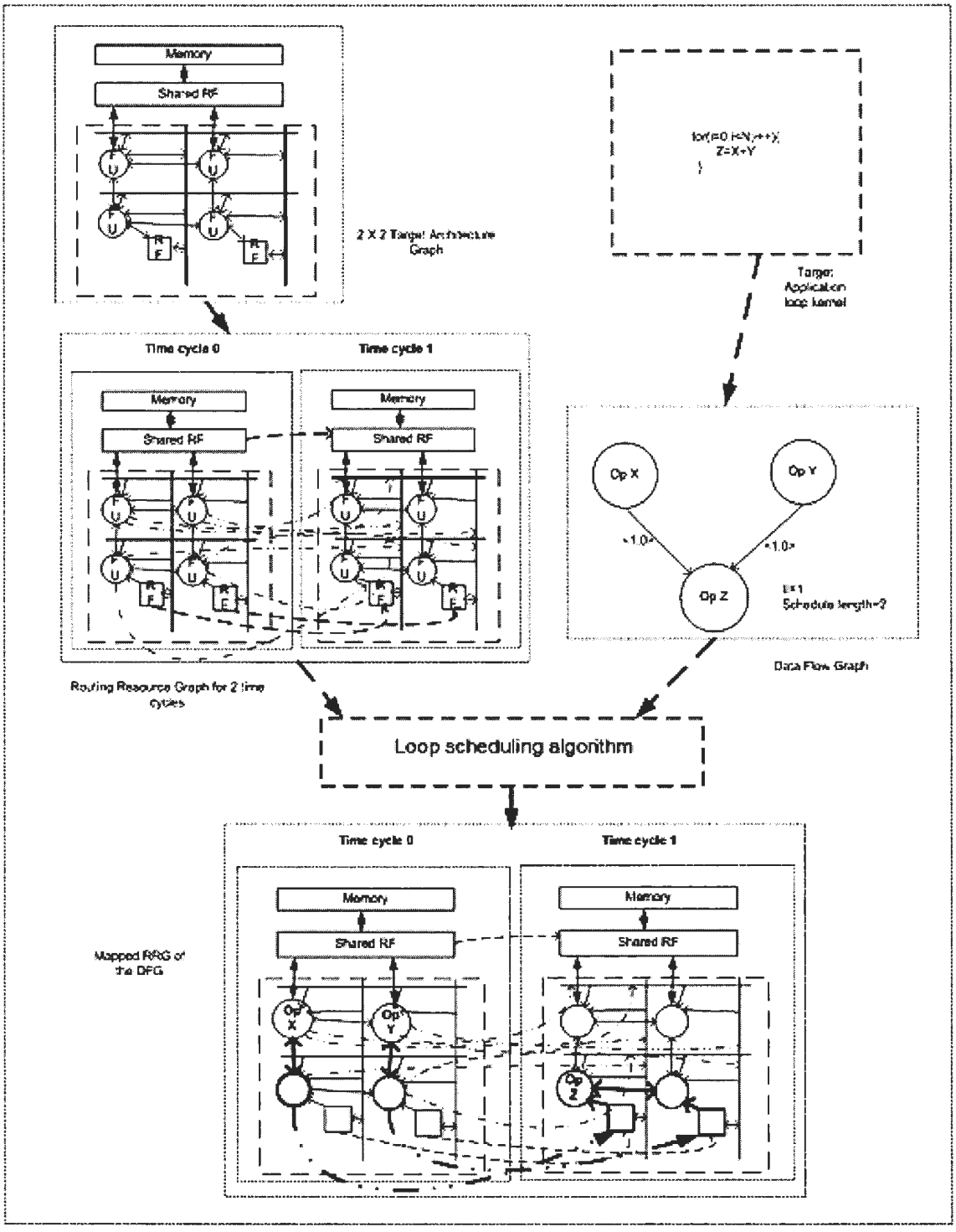


Figure 2.11: Overall mapping of loop kernel of DFG onto RRG of CGRA

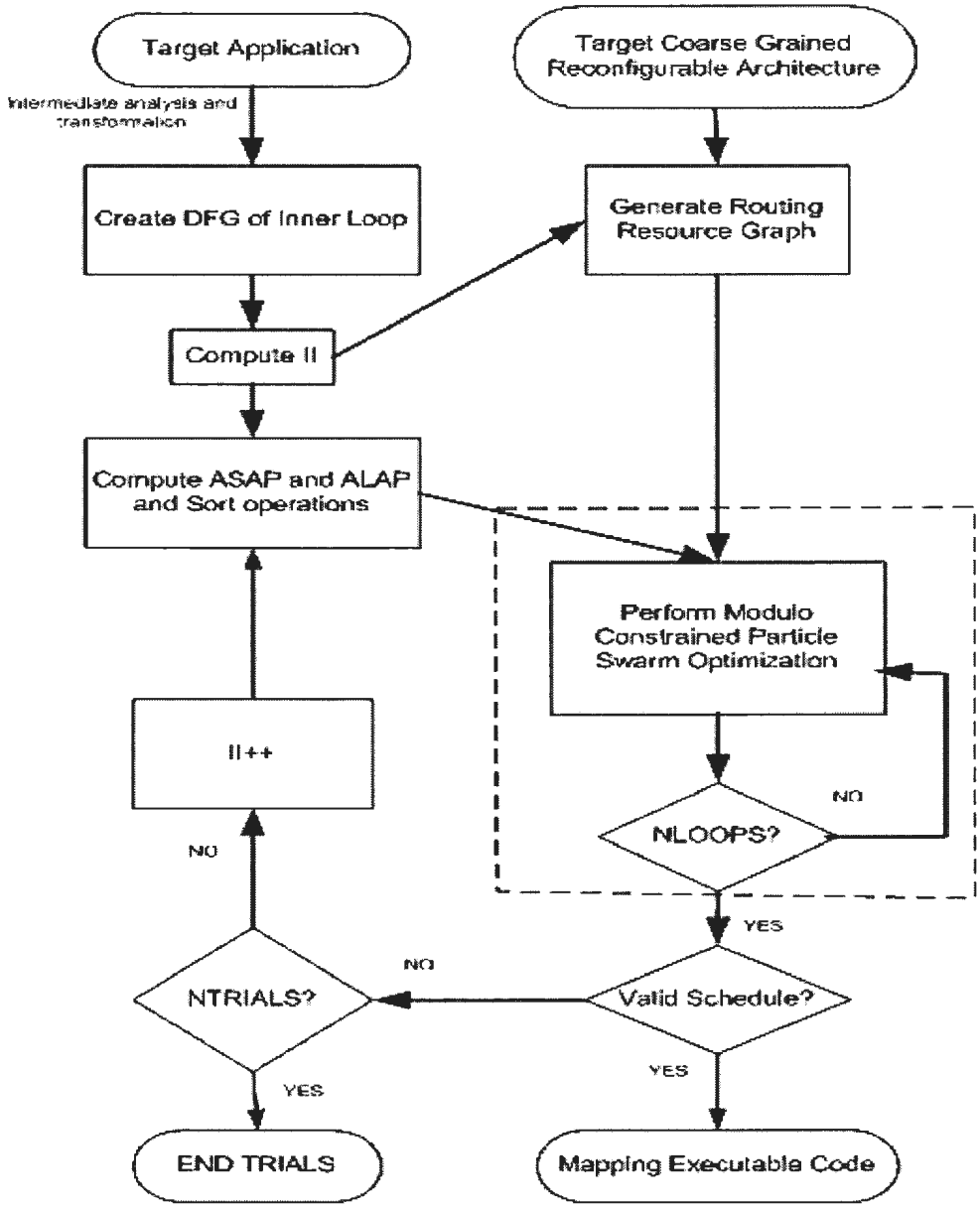


Figure 2.12: Compilation flow of the proposed algorithm

```
Procedure ModuloSch_Place_Route (DFG, TA)
begin
    II := MII (DFG)
    dfgList := ComputeASAPandALAP (DFG)
    sortedDFG := sort (dfglist)
    max_schLength := findschLength(sortedDFG)
    schSucess := false
    NTRIALS := max_schLength - II
    trials :=II
    while !schSucess&& trials<NTRIALS do
        CreateRRG(TA, II, max_schLength)
        schSucess := MCHPSO(sortedDFG, RRG, II, max_schLength)
        II++
        trials++
    end while
end
```

Algorithm 2.0: Mapping DFG onto RRG

$$newPcoord_i = \text{mutationOperator}(avail\_slots, DFG\_Rop, P_{new}) \tag{2.7}$$

where

- flip1, flip2 are random boolean variables to select the particle's new position (P_{new}) either from global best position (P_{Gbest}) or from the local best position P_{Lbest} position or the current particle position $currentP_i$.
- Random_op selects 1 of the data flow graph nodes from P_{new} and saves it in the DFG_Rop variable.
- MRT_check finds the FU slots that are direct neighbors of DFG_Rop position which are free in the TA of P_{new}'s MRT. MRT_check saves the list of slots in the $avail\_slots$ variable.
- mutationOperator mutates the position of the random operation to any 1 of the $avail\_slots$. Now $newPcoord_i$ contains the new particle position for the next

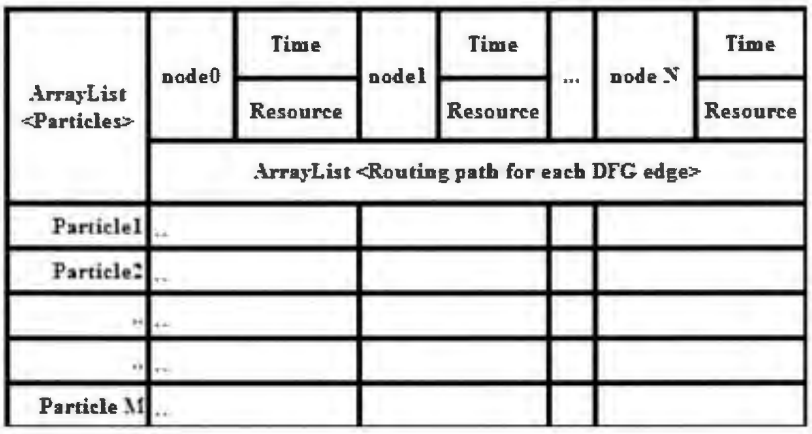


Figure 2.13: Particle encoding for scheduling

iteration.

Once the mutation is done on the particle, then the new particle coordinates are ready for the next generation of MCHPSO. The mutationOperator helps the particles to explore more solutions instead of getting caught in a local optimum. The inner loop of MCHPSO to find the best solution continues until a given number of iterations are completed. The best mapping schedule solution goes through a validity checker for overuse of resources, routability of all edges and maintenance of modulo constraints.

In the standard PSO, a large inertia weight ω facilitates a global search while a small inertia weight facilitates a local search. In the MCHPSO system, the inertia weight is taken care by 1 of the fitness values called overuse. In the initial population, particles explore more with large overuse. As the iteration proceeds, particles exploit the solution with less overuse. When the particles start to perform more of a local search, mutation operator helps them not to get them caught at a local optimal solution. In the MCHPSO, there is no usage of maximum velocity parameter. Instead

```
Procedure MCHPSO (sortDFG, RRG, II, schLength)
begin
    for each operation in sortDFG do
        Initialize Particles
        Initialize MRT(#FUs,II)
    end for
    repeat NLOOPS times
        for each particle in Particles do
            Find the fitness value from GetRoutingCost (RRG, particle)
            if the fitness value is better than the best fitness then
                Set current fitness value as the new particle best fitness
            end if
        end for
        Find the global best particle
        for each particle do
            Calculate the new particle position according to the Equations 2.4, 2.5,
2.6, and 2.7
            Update particle search position
        end for
    end repeat
    if validSchedule (bestparticle) then return true
    else return false
    end if
end
```

Algorithm 2.1: The MCHPSO algorithm

the particles stay within the size of routing resource graph size during placement, schedule and route. In the MCHPSO, there is no usage of c_1 and c_2 values. In the preliminary investigation of PSO, usage of c_1 and c_2 did not help the particles to modulo schedule, so they are not used in the update.

2.2.2.0 Need for the mutation operator

In modulo constrained particle swarm without a mutation operator, we found particles stay in the same solution for a long time in some iteration. When there was no mutation operator in that execution, the particles could not come out of that local

optimum to find a valid solution. Mutation operator is needed to avoid local minimum because as the iteration increases the particles tend to get close to each other and can get caught in a local solution. Mutation operator helps the particles to perform more local search when they are closer to the solution.

MCHPSO does not use velocity, or ω or c_1 or c_2 to update particles position. Instead the particles get updated to search near the current position or a local best position or a global best position with the help of a mutation operator.

2.2.3 Fitness Calculation

The pseudocode of the fitness calculation is given in Algorithm 2.2. The fitness calculation algorithm (*GetRoutingCost*) considers multiple objectives from the routing paths produced by Dijkstra's shortest-path algorithm (*i.e, the getShortestPath method in the algorithm*) [Dijkstra, 1959]. The 3 main objectives considered in this work are that no resource in the TA is overused, all edges in the DFG are routable, and fewest resources are used to route. The routing cost is computed by accumulating the cost of all used RRG nodes incurred by the placement and routing of all the edges.

In every iteration, each particle's fitness value (say p) is compared with its local best fitness value (say q), from the previous iteration. If p's number of routable edges value is greater than q, then p is chosen else q is chosen. If still both p and q are the same then check if p's number of overused resources is lesser than q, then p is chosen else q is chosen. If still both p and q are same then check if p's total routing cost is less than q, then p is chosen else q is chosen. If p's values are chosen then the local best position of the particle is updated with the current position values. Similarly,

the above comparison is done for each particle's local best fitness values (say p) with the global best fitness values (say q), from the previous iteration. The global best position is updated based on the best particle's local best position.

Each node in the RRG has a capacity, base cost [Mei *et al.*, 2003a], availability, and number of times used. Majority of RRG nodes, have a capacity of 1 whereas a few types of nodes such as register files have a capacity larger than 1. The *Findroutingcost* method calculates the usage of each resource in the routing and also calculates if a resource is overused that its capacity (*findPathoverused*).

```
Procedure GetRoutingCost(RRG,psoPart)
    begin
        rcost:=0
        notRoutableEdges:=0
        overusedNodes:=0
        edgeSet:={Scheduled and Placed PSOparticle}
        for each edge e in edgeSet
            u:=e.source
            v:=e.target
            path:=getShortestPath(u,v)
            if(path ≠ NULL) then
                rcost+=Findroutingcost(path);
                overusedNodes+=findPathoverused(path);
            else
                notRoutableEdges++
            endif
        endfor
        return (rcost, notRoutableEdges, overusedNodes)
    end
```

Algorithm 2.2: Routing cost fitness value for MCPSO

2.2.4 Configuration File and Final Schedule

Once the MCHPSO algorithm is completed, it generates the final schedule of one iteration such that the modulo constraints and dependence constraints are met. The MRT generated for the final schedule, produces a configuration text for each time cycle. The configuration text contains the operation for each FU of the TA, reservation of routing resources and the memory unit operations in each cycle. An example final schedule is shown in Table 2.1 and for the DFG, in Figure 2.9.

Table 2.1: Final schedule result of the DFG onto the TA

Resources/ Schedule Length	F1	F2	F3	F4	RF1	RF2	RB1	RB2	CB1	CB2
0	2	c1	2-x	c1-x						
1	4	c2	Op x	c1-x	r0:2-x	r0:c1-x				
2		Op y			r1:x-z		4-y			c2-y
3					r1:x-z					y-z
4					r1:x-z	r1:y-z			x-z	
5	Op z		y-z	y-z	r2:x-z	r1:y-z			x-z	

2.3 Final schedule of the MCHPSO Algorithm

To evaluate MCHPSO algorithm, a slightly modified architecture from ADRES [Mei *et al.*, 2005a] was used. Various digital signal processing (DSP) benchmarks [Texas Instruments. inc, 2009], [Texas A&M University-Kingsville, 2009], [University of Patras, 2009] were used to evaluate the performance of the MCHPSO algorithm. The implementation of MCHPSO algorithm is written in Java. The TA and loop body description are given in files to the proposed algorithm. The evaluation is done to check whether MCHPSO was able to solve intra-dependent inner loop body mapping

onto the CGRA with a lower II. The routing algorithm gives the fitness value in 3 different styles:

0. Routable Edges.

1. Overuse of Resources

2. Total cost of Routing.

The particle holding the maximum routable edges with no overuse of resources and minimum cost is taken to be the best particle. The reliability and performance of MCHPSO algorithm is tested with more experiments on varying interconnection topologies, memory ports and distributed register files. To find the suitability and effectiveness of MCHPSO algorithm, it is compared with various other modulo scheduling algorithms and heuristic methods such as modulo scheduling, with simulated annealing [Vassiliadis and Soudris, 2007b], and memory conscious modulo scheduling [Dimitroulakos *et al.*, 2007].

2.4 Conclusion

This chapter discussed the modulo constrained, hybrid particle swarm optimization algorithm to solve the scheduling problem to map a DFG of loop body onto the TA graph. The MCHPSO algorithm, with the combination of PSO and mutation operator, was discussed to map effectively the given target application loop onto the CGRA. The extensions, evaluation and applications of the MCHPSO algorithm were also discussed. A detailed analysis of MCHPSO algorithm will be discussed in the next chapter.

Chapter 3

Performance Analysis of MCHPSO Algorithm

3.0 Introduction

In this chapter, the performance data of the modulo-constrained hybrid particle swarm optimization (*MCHPSO*) is discussed. The proposed algorithm is designed to solve the problem of mapping a Data Flow Graph (*DFG*) for a loop body in the application onto the resource and routing graph (*RRG*). The MCHPSO algorithm has been explained in Chapter 2. The MCHPSO algorithm, effectively maps with the combination of Particle Swarm Optimization (*PSO*) algorithm and a mutation operator. The results obtained from the analysis of the work are discussed in the following sections. These results help us to understand the research problem and to extend the algorithm to map loops with different characteristics, as discussed in Chapters 4 and 5.

3.1 Analysis of Scheduling

Scheduling a loop onto the coarse-grained reconfigurable architectures (*CGRA*) consists of 3 main parts:

0. Placement of each operation of the DFG onto the CGRA computing resource, FU;

1. Scheduling the execution time of each operation of the DFG;

2. Routing every edge in the DFG as a path in the RRG.

The most important constraint in a scheduling algorithm is getting a valid result for the schedule, with no interference among the placed, scheduled, and routed resources. When an architecture is considered, the constraints in the architecture such as interconnection topologies, and availability of computing and memory resources play a major role in finding a schedule for the loop kernel. Therefore, different architecture parameters were tried so that the performance of the algorithm could be tested on a number of architectures. The number of nodes and edges in the DFG determine the complexity of the kernel to be mapped onto the CGRA. The usage of resources, mapping time, and schedule density were estimated to analyze the performance of mapping algorithm.

3.2 Modulo Scheduling with MCHPSO

3.2.0 Experimental Set Up

The MCHPSO scheduling algorithm was written in Java and executed on an Intel Core 2 Duo CPU with 4 GB RAM and a clock speed of 2 GHz. To schedule a loop onto the CGRAs, 2 main inputs were required for the MCHPSO scheduling algorithm. The first input is the DFG generated from the benchmark loops. The loop extraction process is described in Chapter 1. The second input for the MCHPSO is the CGRA architecture. The target architecture (TA) graph is created from the TA configuration file as described in Chapter 2. An example of DFG generation is discussed in the next subsection.

Other than the 2 main inputs, DFG and TA, MCHPSO requires the following parameters: the number of particles is 10, the relax-factor is the II, the number of trials for each initiation interval (II) is the difference of schedule length and II, and the number of iterations per trial is 20. A relax-factor is used to adjust the as late as possible ($ALAP$) values, when the scheduler finds difficulty in a very tight range to start the next iteration while finding a place for leaf nodes.

3.2.0.0 DFG Generation

One of the main inputs to the MCHPSO scheduling algorithm is the DFG generated from the application loop kernel. The generation of the DFG and its parameters are described in Chapter 2. The benchmarks taken for the algorithm are shown in Table 3.0. The first 4 benchmarks were derived from the C reference code of Texas Instruments (TI) Inc. [Texas Instruments. inc, 2009]. The next 2 benchmarks

Table 3.0: DFG characteristics of the benchmarks

Benchmarks	# Nodes	# Edges	MII	Initial Schedule length
8X8 IDCT_hor	78	108	3	9
4X4 FFT	67	107	3	10
8X8 FDCT_hor	74	102	4	8
8X8 FDCT_ver	73	100	3	8
latasynth	20	20	1	8
latanal	20	21	1	5
FIR_cplx	25	33	2	6
Volterra	28	35	2	5
IIR	36	51	2	10
IIR_biquad	35	36	3	8

were based on lattice filter [Texas A&M University-Kingsville, 2009]. The last 4 benchmarks were taken from [University of Patras, 2009],written by the authors of [Dimitroulakos *et al.*, 2007].

We will now consider the lattice synthesis filter benchmark as an example to create a DFG. The lattice synthesis filter application code is shown in Figure 3.0. The benchmark was analyzed to find the inner loop body (Figure 3.0) of the application code. A description file of the loop kernel is created as shown in Figure 3.1. From the description file, the DFG created for the loop kernel of the application is shown in Figure 3.2. Once the DFG is created, it is ready to be passed to the scheduling algorithm to be mapped onto the RRG.

```
// Lattice LPC Synthesis Filter
q = length(residu);
N = 250;
betold = zeros(1,P1);
for seg = 1:q/N,
  s = xc((seg-1)*N+1:seg*N);
  K = trans((seg-1)*P+1:seg*P);
  for n=1:N,
    eps(P1) = s(n);
    for i=P1:-1:+2
        eps(i-1) = eps(i) + K(i-1)*betold(i-1);
        bet(i) = betold(i-1) - K(i-1)*eps(i-1);
    end
    bet(1) = eps(1);
    betold = bet;
    x((seg-1)*N+n) = eps(1);
  end          // End of the synthesis filter
end
```

Figure 3.0: Lattice synthesis filter code

3.2.0.1 TA Graph Generation

The TA graph was generated from the configuration shown in Table 3.1, along with the interconnection between the resources specified in the architecture connection file. To have a comparable architecture with other works and rich interconnections, an 8×8 CGRA array were employed. The 8×8 CGRA array comprises of Meshplus1 FU topology, row and column buses, private RFs that connects with diagonally adjacent register files (RFs) and shared RF (SRF) for the memory unit (MU) connections. The private RFs could only handle data and not predicates. The number of resources and topology are similar to the work reported in [Vassiliadis and Soudris, 2007a]. The RRG was generated by replicating the TA for each cycle, along the time axis until the maximum schedule length was reached. The edges of the RRG are explained in

```
17          //No of nodes in DFG
//Nodename-//#incomingedges,#outgoingedges:
//Outgoingedgenodename1,Outgoingedgenodename2,..
i-0,3:im1,epsi,beti
1-0,1:im1
eps-0,2:epsi,epsim1
k-0,1:kim1
betold-0,1:betoldim1
bet-0,1:beti
im1-2,3:epsim1,kim1,betoldim1
epsi-2,1:epsim1new
mulkbetold-2,1:epsim1new
epsim1new-2,1:epsim1
epsim1-2,1:mulkeps
kim1-2,1:mulkbetold,mulkeps
betoldim1-2,1:mulkbetold,betnewi
mulkeps-2,1:betnewi
betnewi-2,1:betout
beti-2,1:betout
betout-2,0:
```

Figure 3.1: DFG description file for the Lattice synthesis filter in Figure 3.0

Chapter 2.

3.2.1 Scheduling Results

The MCHPSO algorithm takes the loop kernel and a CGRA architecture as input. When the DFG of the loop kernel is read, the particles in MCHPSO generate a partial place and schedule result. Each node in the DFG is mapped onto the computing resources of the RRG such as the FUs. The FU number for each operation in the

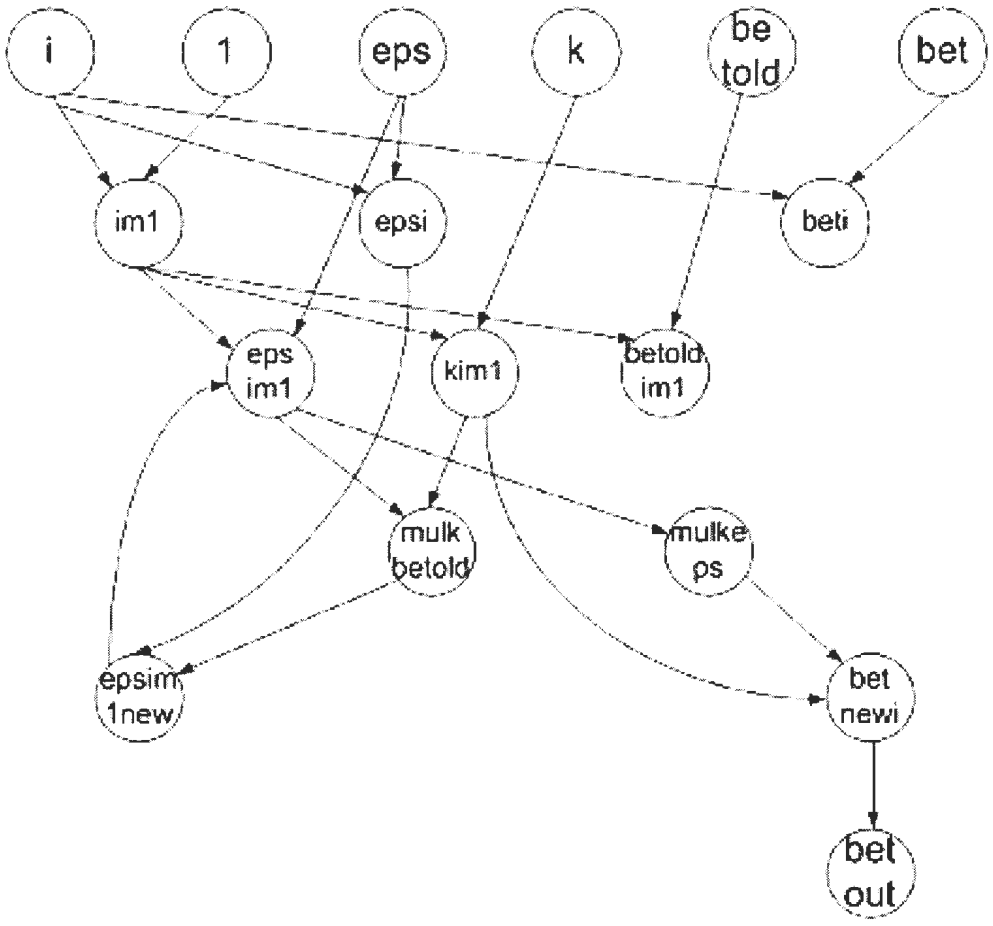


Figure 3.2: DFG corresponding to the code in Figure 3.0

DFG shows where the operation will be executed. The time at which the placed operation will be executed is shown in the schedule time. All the particles, with schedule and place values, go through the router to discover how many edges are routable. The best particle is chosen based on the fitness constraints explained in Section 2.2.3 of Chapter 2. The algorithm stops once all the edges are routable and there is no overuse of resources.

A schedule and place schedule result for the DFG in Figure 3.2, is shown in Table

Table 3.1: 8 X 8 CGRA configuration

Resources	Total numbers	Capacity	Reads	Writes
FU	64	1	N/A	N/A
RF	56	8	8	8
SRF	1	16	8	8
CB	8	1	N/A	N/A
RB	8	1	N/A	N/A

3.2. The path of each edge, routable from the routing algorithm of the DFG, is shown in the routing result displayed in Table 3.3 and Table 3.4, where the first column shows the edge number and the second and fourth columns show the name of the source and target DFG operation. The path between the mapped source and target DFG operations are shown in the third column. The notation for the paths are explained as follows: F represents the FU, R represents the RF, CB represents the column bus and RB represents the row bus. The cost of how much resources are needed for routing of each path is shown in the fifth column.

Each schedule, place, and routing result denotes a particle state. For every iteration, each particle finds a scheduling result for the given graphs. The fitness result of every iteration of the particles is shown in Figure 3.3. In each iteration, particles with fitness of maximum edges routable with no overuse is shown. The particle with the highest fitness is chosen as the best particle for each iteration. The other particles move towards the best particle in the next iteration. The role of mutation operator in each iteration gives randomness to each particle and enables them to try for the best position. At the end of iteration 66, particle 5 finds all the edges (i.e. 108) to be routable in the DFG and gives the best scheduling result possible. Every particle saves its best local fitness from all the iterations in a local-best fitness vector which is

Table 3.2: Scheduled and placed results of the lattice synthesis loop kernel

DFGname	SchTime	PlacedFU
bet	0	1
betold	0	18
k	0	3
eps	0	4
1	0	17
i	0	19
im1	1	37
kim1	2	58
betoldim1	2	50
epsi	1	39
mulkbetold	3	59
epsim1new	4	43
epsim1	5	33
mulkeps	6	47
betnewi	7	46
beti	1	12
betout	9	20

depicted in Figure 3.5. Some particles are penalized if they cannot find a valid route by a fitness of 0 and that is shown as the missing particles in the Figure 3.5. From the local-best fitness vector of every particle, a global best particle is chosen, depending on the fitness value found. The best fitness value found for selected iterations is shown in Figure 3.4.

Table 3.3: Routing results of lattice synthesis loop kernel -part1

Edge no	SourceDFG name	Path from Source-Target in RRG	TargetDFG name	Path cost
0	bet	F1time-0->F1temptime-0->R2time-1->F11time-1->F11temptime-1->F12time-1	beti	4
1	betold	F18time-0->F18temptime-0->R15time-1->C32time-1->R47time-2->F50time-2	betoldim1	4
2	k	F3time-0->F3temptime-0->R2time-1->CB2time-1->R14time-2->C22time-2->R50time-3->R50time-4->R50time-5->R50time-6->R50time-7->R50time-8->R50time-9->R50time-0->R50time-1->R50time-2->F58time-2	kim1	15
3	eps	F4time-0->F4temptime-0->C01time-0->R28time-1->F39time-1	epsi	3
4	eps	F4time-0->F4temptime-0->R3time-1->F10time-1->F10temptime-1->R9time-2->C11time-2->R25time-3->R25time-4->R25time-5->F33time-5	epsim1	9
5	1	F17time-0->F17temptime-0->C12time-0->F53time-1->F53temptime-1->F40time-1->F40temptime-1->F37time-1	im1	6
6	i	F19time-0->F19temptime-0->R16time-1->C02time-1->R44time-2->CB5time-2->R26time-3->R26time-4->R26time-5->R26time-6->R26time-7->R26time-8->R26time-9->R26time-0->R26time-1->F37time-1	im1	14
7	i	F19time-0->F19temptime-0->F31time-0->F31temptime-0->F51time-0->F51temptime-0->R46time-1->F57time-1->F57temptime-1->F44time-1->F44temptime-1->F39time-1	epsi	10
8	i	F19time-0->F19temptime-0->R14time-1->F25time-1->F25temptime-1->F12time-1	beti	4
9	im1	F37time-1->F37temptime-1->R29time-2->R29time-3->R29time-4->R29time-5->F34time-5->F34temptime-5->F33time-5	epsim1	7
10	im1	F37time-1->F37temptime-1->CB6time-1->R45time-2->F58time-2	kim1	3

Table 3.4: Routing results of lattice synthesis loop kernel -part2

Edge no	SourceDFG name	Path from Source-Target in RRG	TargetDFG name	Path cost
11	im1	F37time-1->F37temptime-1->F42time-1->F42temptime-1->R31time-2->F36time-2->F36temptime-2->F49time-2->F49temptime-2->F50time-2	betoldim1	8
12	kim1	F58time-2->F58temptime-2->R50time-3->F59time-3	mulkbetold	2
13	kim1	F58time-2->F58temptime-2->R53time-3->CB8time-3->R39time-4->R39time-5->R39time-6->F47time-6	mulkeps	6
14	betoldim1	F50time-2->F50temptime-2->R47time-3->F59time-3	mulkbetold	2
15	betoldim1	F50time-2->F50temptime-2->R47time-3->F60time-3->F60temptime-3->CB7time-3->R35time-4->R35time-5->R35time-6->R35time-7->F46time-7	betnewi	9
16	epsi	F39time-1->F39temptime-1->R31time-2->R31time-3->R31time-4->F43time-4	epsim1new	4
17	mulkbetold	F59time-3->F59temptime-3->F64time-3->F64temptime-3->F61time-3->F61temptime-3->F48time-3->F48temptime-3->R40time-4->F43time-4	epsim1new	8
18	epsim1new	F43time-4->F43temptime-4->R30time-5->F33time-5	epsim1	2
19	epsim1	F33time-5->F33temptime-5->F45time-5->F45temptime-5->R34time-6->F47time-6	mulkeps	4
20	mulkeps	F47time-6->F47temptime-6->R34time-7->F46time-7	betnewi	2
21	betnewi	F46time-7->F46temptime-7->CB1time-7->R6time-8->F5time-8->F5temptime-8->SR1time-9->F20time-9	betout	6
22	beti	F12time-1->F12temptime-1->R11time-2->CB4time-2->R23time-3->R23time-4->R23time-5->R23time-6->R23time-7->R23time-8->R23time-9->F28time-9->F28temptime-9->F20time-9	betout	12

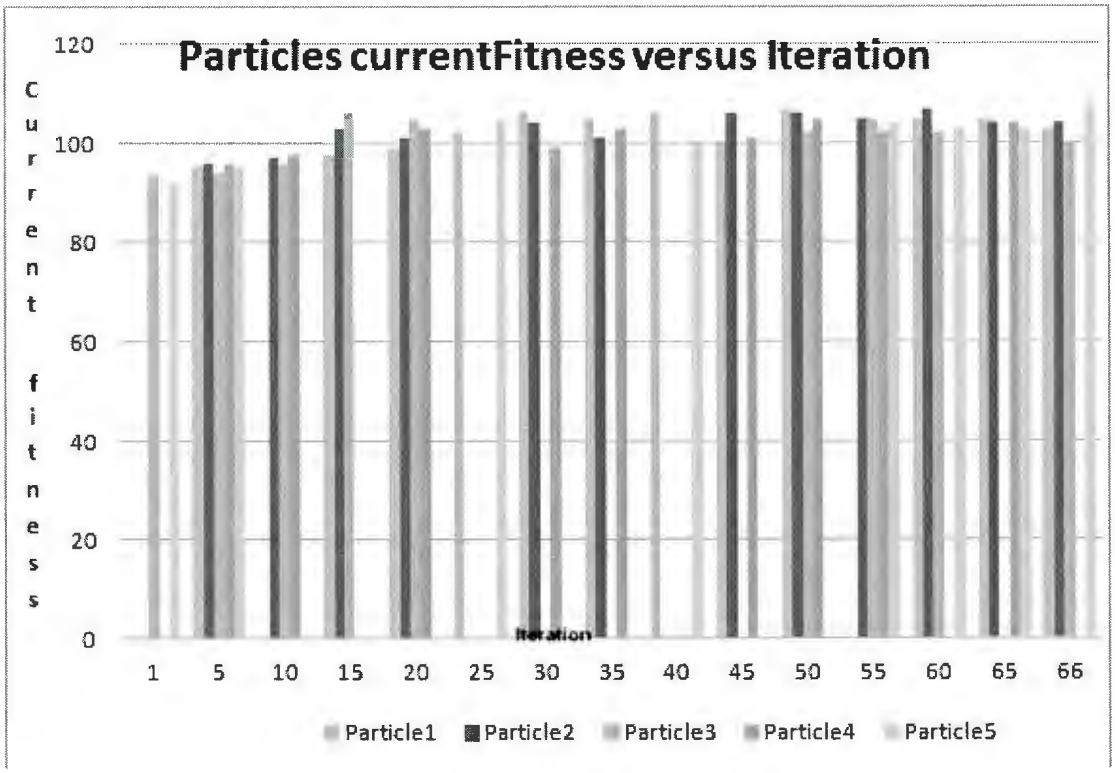


Figure 3.3: All particles currentFitness versus Iteration

3.2.2 Mapping of Nodes and Routing of Edges

The MCHPSO was experimented on an 8×8 CGRA configuration with FU that can either place or route as well as on an 8×8 CGRA configuration with FU reuse. The schedule, place, and route results from MCHPSO of all the selected benchmarks on an 8×8 CGRA configuration with FU reuse are shown in Table 3.5. The first column shows the benchmark name, second column denotes the number of operations in the loop kernel, and the third column shows the initiation interval at which the loop kernel is mapped. The fourth column shows the instructions per cycle (IPC) which is calculated by

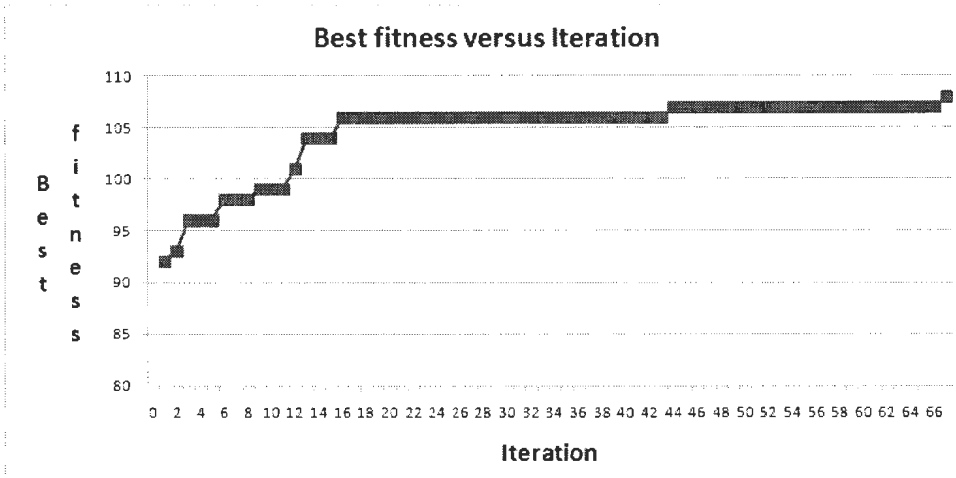


Figure 3.4: Global best fitness for every iteration

$$\text{IPC} = \frac{\text{N\_Instruction}}{\text{II}} \tag{3.0}$$

the Equation (3.1).

The schedule density, without routing, considers the number of FUs used in the placement. The schedule density, with routing, considers the count of FUs used in the placement as well as in routing of edges. The fifth column shows the schedule density without routing and the sixth column shows the schedule density of FU, with routing, which are calculated as follows

$$\text{schDensity\_NO\_R} = (\frac{\text{IPC}}{\text{number of FU}}) * 100 \tag{3.1}$$

$$\text{schDensity\_WR} = \text{no of stages} * (\frac{\text{N\_Instruction} + \text{FU used in routing}}{\text{number of FU in RRG}}) * 100 \tag{3.2}$$

where,

- schDensity_NO_R: Schedule density of the FUs with only placement.

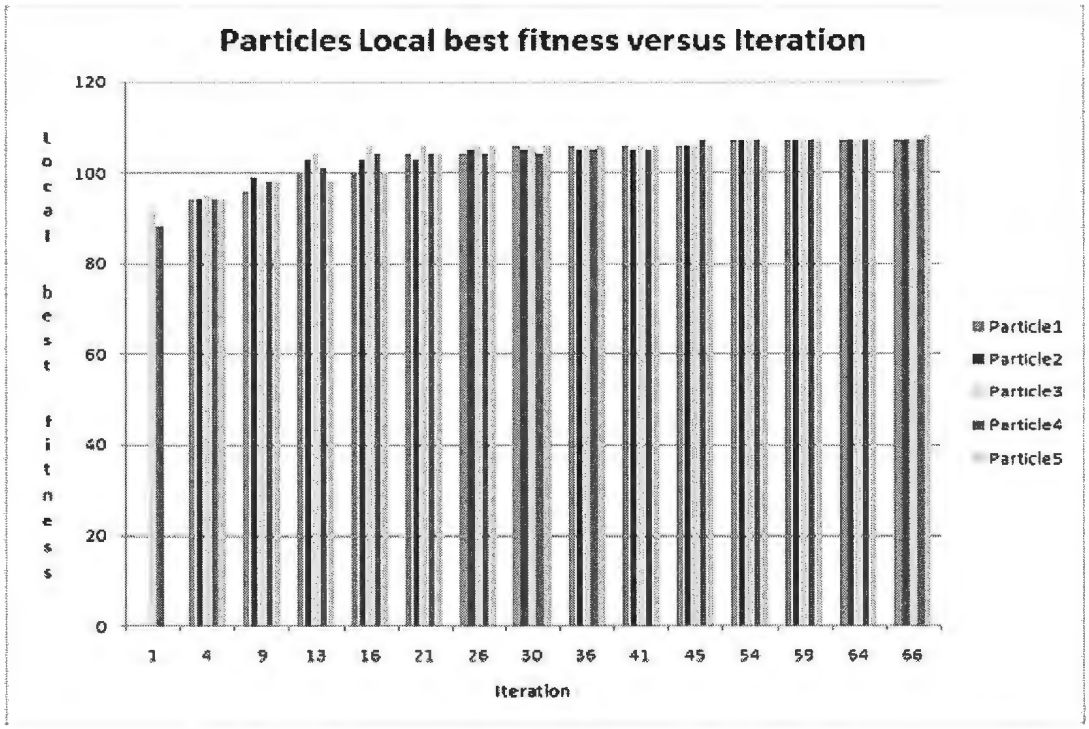


Figure 3.5: BestFitness of all particles versus Iteration

- schDensity_WR : Schedule density of the FUs with routing.
- N_Instruction : Number of Instructions in the DFG.

The eighth column shows the number of stages overlapped, which is calculated as number of stages $= \lceil \frac{\text{Schedule Length}}{\text{II}} \rceil$.

The seventh column shows the total CGRA usage percentage, including all the computation and routing resources in the CGRA such as FU, RF, CB. RB and SRF used for the scheduling of loop kernel. The total CGRA utilization percentage is calculated by

$$\text{Total\_Util} = \text{number of stages} * (\frac{\text{N\_Instruction} + \text{totalResUsed}}{\text{RRG size}}) * 100 \qquad (3.3)$$

where,

- N_Instruction : Number of Instructions in the DFG
- totalResUsed : Total RRG resources used in the routing path.

$$\text{resUsedPercentage} = \text{number of stages} * (\frac{\text{usedRes}}{\text{availRes}}) * 100 \qquad (3.4)$$

where,

- resUsedPercentage : Percentage of resources used with overlap.
- usedRes : Number of resources of the particular type such as FU, RF, CB, RB, and SRF used in routing.
- availRes : Available resources of the particular type in the RRG.

The last column shows the time taken in seconds to schedule the loop kernel. The mapping results show that the proposed scheduling algorithm MCHPSO utilizes from 31.25% to 79.69% of the total FUs available in the CGRA. The FU usage depends on the size of the DFG and the number of stages of the loop. The largest loop kernels, such as IDCT_hor (horizontal pass) and FFT, are scheduled within a maximum of 105.89 seconds. The time to schedule a loop kernel depends on the size of DFG, II and the modulo constraints. The larger the loop, the higher the constraints on resources and longer the time the algorithm takes to complete the mapping process.

Table 3.5: Overall mapping results of the DSP benchmarks in 8 x 8 CGRA

Benchmarks	# ops	MII	II	OPC	Schedule Density (without routing)	Schedule Density (with routing)	Total CGRA Util %	# Stages	Exe Time in Seconds
FIR_complex	25	2	2	12.5	18.75	39.06	12.59	4	8.72
Lattice synth	20	1	1	20	29.69	79.69	22.06	10	12.58
Volterra	28	2	2	14	21.88	34.38	14.06	3	6.87
IIR	36	2	2	18	28.13	62.5	21.14	4	12.55
IIR_biquad	35	3	3	11.7	17.19	31.25	9.25	4	16.93
8X8 IDCT_hor	78	3	3	26	40.63	73.44	29.47	5	93.11
4X4 FFT	67	3	3	22.3	34.38	75.52	29.66	5	105.89
8X8 FDCT_hor	74	4	4	18.5	29.69	63.28	18.34	3	27.01
8X8 FDCT_Ver	73	3	3	24.3	37.5	78.13	21.2	4	55.67

Experiments show that the MCHPSO algorithm could handle a wide range of loops with different number of operations.

The MCHPSO was experimented on an 4×4 CGRA configuration with FU that can either place or route as well as on an 4×4 CGRA configuration with FU reuse. The schedule, place, and route results from MCHPSO of all the selected benchmarks on an 4×4 CGRA configuration with FU reuse is shown in Table 3.6. The first column shows the benchmark name, the second column denotes the number of operations in the loop kernel, and the third column shows the initiation interval at which the loop body is mapped. The fourth column shows the schedule density without routing, as calculated by the Equation (3.1). The schedule density, without routing, considers the count of FUs used in the placement. The fifth column shows the schedule density of FU, with routing, as calculated by the Equation (3.2). The last column shows the

Table 3.6: Overall mapping results of the DSP benchmarks in 4 x 4 CGRA

Bench-Marks	# Ops	MII	II	ScheduleDensity (without routing)	ScheduleDensity (with routing)	Time in Seconds
FIR_cplx	25	3	3	50	68.75	0.84
latasynth	20	2	2	56.25	78.13	0.66
latanal	20	2	2	56.25	68.75	0.53
Volterra	28	4	4	43.75	57.81	1.36
IIR	36	4	4	56.25	78.13	2.17
IIR_biquad	35	5	5	43.75	61.25	1.77
8X8 IDCT_hor	78	6	7	68.75	89.29	7.2
4X4 FFT	67	5	7	56.25	81.25	9.86
8X8 FDCT_hor	74	7	7	68.75	90.18	6.45

execution time taken in seconds on an Intel Pentium M with 1 GB RAM and a clock speed of 1.73 GHz.

From the mapping results, it is clear that the higher the number of loop operations, the larger the routing resources required. Our MCHPSO scheduling algorithm was able to map the benchmarks, for both the 4×4 and the 8×8 CGRA configurations. The II achieved to map the benchmarks were the minimal II in most cases, and close to the minimal in others.

3.2.3 Analysis of Functional Units Usage for Different Topologies

The various topologies of FU are explained in Section 2.1.1.1 of Chapter 2. In this section, the flexibility of each topology and its usage are discussed. The interconnection topologies are (1) a mesh based architecture of 4 neighboring FU connections;

(2) a meshplus1 architecture of 8 neighboring FU connections; and (3) a meshplus2 architecture of 4 neighboring FU connections along with every FU connected with all other FUs in the same row and the same column (please refer to Figure 2.2 of Chapter 2). Table 3.7 shows the comparison of Functional Unit usage using various topologies. This experiment is done on a 4×4 CGRA. The first column shows the 2 benchmarks taken for comparison. IDCT_hor and FFT benchmarks were chosen because they did not schedule with the minimal II. The FU usage of the mapped II schedule is compared with the previous initiation intervals (like $II - 1, II - 2$). The second column shows the minimal II. The third column shows the II achieved to find a schedule without any overuse of resources. The fourth column shows the percentage of FU usage, considering only the placement. The fifth, sixth seventh, eighth columns show the FU usage after scheduling, placement, and routing in mesh, meshplus1, meshplus2 and star topologies. The topologies which overuse in scheduling, placement, and routing have more than 100% usage. From row1, row3 and row4, it shows that the overuse of FUs is reduced when the interconnections were increased. Maximum FU utilization is achieved in the case of mesh topology. When the interconnections are increased in the other topologies, the utilization of same FUs is reduced and other FUs are explored and used. When a benchmark has a lot of edges to route, the flexible interconnection helps the MCHPSO scheduling algorithm to achieve a valid schedule, with no overuse of resources.

Table 3.7: Usage of Functional Units with various topologies

				Mesh	MeshPlus1	MeshPlus2	MeshPlus2 and Star
Bench-Marks	Min II	II	P	P&R	P&R	P&R	P&R
8X8 IDCT_hor	6	6	81.25	112.5	107.29	109.38	101.04
		7	68.75	91.96	90.18	90.18	89.29
4X4 FFT	5	5	81.25	128.75	128.75	122.50	120.00
		6	68.75	105.21	105.21	102.08	104.17
		7	56.25	85.714	78.571	83.928	85.714

3.2.4 Analysis of Register Files Usage with Different Interconnections

The usage of registers in the RFs was studied, with different numbers of RFs and their interconnections. The various interconnections are (1) each FU having its own private RF; (2) each RF is shared by the FUs in the top and bottom row of the same column; (3) each FU has a RF and the RF is shared among FUs adjacent in all the diagonal directions, as shown in Figure 2.3 of Chapter 2. Figure 3.6 shows the usage of registers for the various register file topologies.

This experiment was done on a 4×4 CGRA with each register file having 4 registers, 4 read ports and 4 write ports. The percentage of register usage with corresponding benchmarks are shown in the graph. When the register usage is above 100%, it is considered as an overuse of registers, and which will not produce a valid schedule. The highest of overuse of registers is found in dedicated RF topology. The shared 4 RFs topology uses the limited number of registers efficiently, but for large benchmarks such as the last two, 8×8 FDCT_hor and 4×4 FFT, it overuses the registers by nearly 20% to 100%. The shared 12 RFs topology utilizes the registers

efficiently when compared with dedicated RF topology. Therefore, the shared 12 RFs topology works the best for all the benchmarks with no overuse of registers.

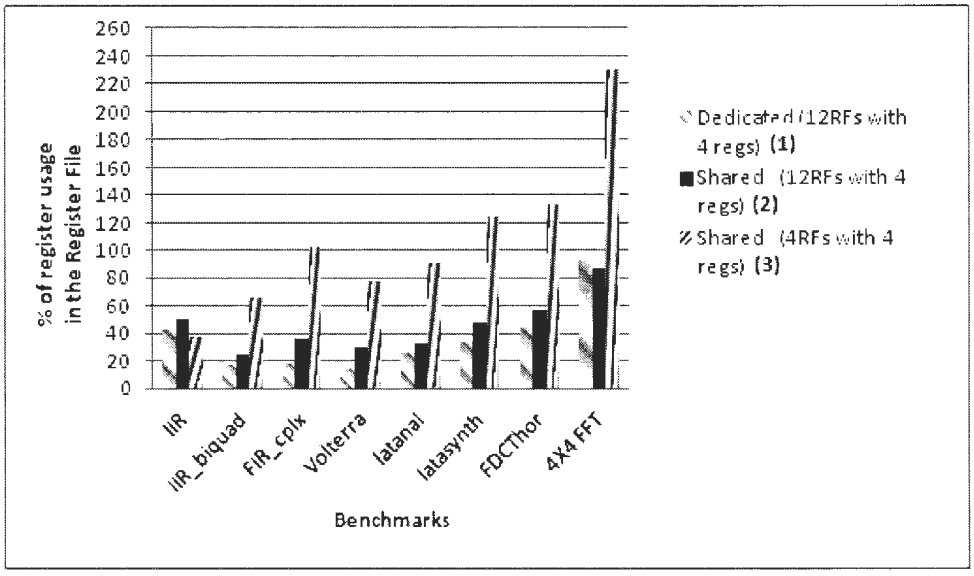


Figure 3.6: Percentage of register utilization in different topology

3.2.5 Effect of Varying Particle Size in MCHPSO algorithm

To determine how many particles should be used in the MCHPSO scheduling algorithm, it was experimented by varying the number of particles used by the algorithm. This experiment was done on an Intel® Core™ i7-860 Processor, with a clock speed of 2.8GHz, using all the 4 cores for an 8x8 simulated CGRA configuration.

The algorithm was not able to come out of the local optimum of the best particle's fitness value when only 5 particles were used. However, a valid schedule was achieved with 10 particles. Table 3.8 shows the comparison of execution time with

Table 3.8: Variation of particle size on an 8 x 8 CGRA

	Execution time (in seconds) of MCHPSO				
Benchmarks	**10**	**25**	**30**	**35**	**40**
8X8 IDCT_hor	22.3	24.0	26.9	31.3	35.5
4X4 FFT	22.0	49.0	48.1	58.6	66.7
8X8 FDCT_ver	12.1	18.7	21.0	24.4	27.8

different particle numbers. The first column shows the 3 large benchmarks taken for comparison. The second to sixth columns show the execution time for particle counts 10, 25, 30, 35 and 40. In all the particle count variations, the MCHPSO algorithm was able to get the valid schedule, with the same usage of resources. The quality of the solution was the same in all the particle size variation. Since there was the same usage in all the different particle counts, it is concluded that 10 particles are sufficient.

3.2.6 Analyzing the Speedup of MCHPSO Algorithm

The Intel Core i7-860 processor (Intel i7-860 processor, 2009) features 4 cores, with a clock speed of 2.8 GHz. It features symmetric multithreading (hyper-threading) so that each core supports 2 threads, for a total of 8 hardware threads. It can run at a maximum clock frequency of 3.46 GHz with Intel Turbo Boost technology. When one core is active, i7 processor operates at a frequency of 3.46GHz. When 2 cores are active, i7 processor operates at a frequency of 3.33GHz. When 3 or 4 cores are active, i7 processor operates at a frequency of 2.93GHz.

To analyze the speedup of our MCHPSO scheduling algorithm, the execution

times of the algorithm were compared, for 1 to 8 processing threads on the quad core processor done in the same environments. Table 3.9 shows the speedup of MCHPSO algorithm on various benchmarks. The first column shows the benchmarks taken for comparison by using logical processors (P) in Intel i7 machine. The second to the ninth columns show the execution time of MCHPSO algorithm. The MCHPSO execution on Intel i7 machine scheduled at the same II as given in Table 3.6. While using 2 processing threads and 2 cores, the speedup was more than 1.5 times than with a single processing thread. While using 4 processing threads, 1 on 4 cores, the speedup is more than 2.5 times than with a single processing thread. While using 8 processing threads on 4 cores, the speedup was more than 3.5 times than with a single processing thread execution. The multithreading, available in the cores, helps the algorithm to process the particle arrays faster. The proposed MCHPSO works faster, with more processing threads. The MCHPSO algorithm did not achieve a lower II than the II given in Table 3.6 in spite of the speedup available by the logical threads. The sublinear speedup was due to the pipelines that don't contend for ALUs, and the memory pipe is to the level 2 cache (the largest cache). Memory contention is probably the most important of those.

3.2.7 Functional Units Capable of Routing and Performing Computations

The computational resources in a CGRA are the functional units, which are capable of executing a set of coarse-grained operations such as add, subtract, multiply, and shift. First, we designed the FUs only to perform computation and to forward in-

Table 3.9: MCHPSO algorithm speed up comparison on an Intel i7 processor

Benchmarks	MCHPSO in Intel® Core™ i7 Processor Execution Time(Seconds)							
	One P	2 P's	3 P's	4 P's	5 P's	6 P's	7 P's	8 P's
FIR_cplx	7.29	4.23	3.1	2.98	2.79	2.68	2.2	2.08
latasynth	6.96	4.17	3.31	3.26	3.2	3.07	2.49	2.43
latanal	2.89	1.76	1.39	1.36	1.3	1.25	1.15	1.06
Volterra	6.26	3.45	2.59	2.36	2.34	2.17	1.86	1.76
IIR	9.13	5.37	3.92	3.65	3.54	3.32	2.81	2.68
IIR_biquad	13.6	7.61	5.4	5.12	5.16	4.56	3.98	3.68
8X8 IDCT_hor	79.31	42.44	32.24	28.82	28.33	27.51	22.69	22.29
4X4 FFT	84.46	44.23	33.16	31.54	29.65	27.58	22.73	22
8X8 FDCT_hor	23.28	13.14	9.97	9.39	8.77	8.41	7.15	6.94
8X8 FDCT_ver	44.28	23.97	18.23	17.12	15.87	15.02	12.3	12.07

Table 3.10: Comparison of FU utilization with placement and routing

Benchmarks	MCHPSO with FU that cannot route if used for execution	MCHPSO with FU that can both route and execute
FIR_cplx	42.19	39.06
Volterra	42.97	34.38
8X8 IDCT_hor	92.19	73.44
4X4 FFT	88.02	75.52
8X8 FDCT_hor	83.98	63.28
8X8 FDCT_ver	88.02	78.13

formation during routing, if they are not performing any operation. Then, the FU was redesigned to have additional ports and switches to perform computation and routing at the same time. The usage of FUs was studied by comparing the 2 different FU configurations, as shown in Table 3.10. The first column shows the benchmarks taken for comparison. The second column shows the percentage of FU usage, with FU configuration that cannot route when it has been used for execution. The third column shows the percentage of FU usage, with FU configuration that can route and execute at the same time. The comparison shows that FU usage decreases when they are capable of both routing and executing and this makes more resources available for mapping larger benchmarks.

3.3 Comparison of MCHPSO with Other Modulo Scheduling Algorithms

Table 3.11 indicates the comparative results of MCHPSO, measured against the modulo scheduling algorithm [Vassiliadis and Soudris, 2007b] used in ADRES, as developed by the IMEC [IMEC, 2009] group. The second column shows the benchmarks used, which are derived from TI Inc. [Texas Instruments. inc, 2009]. The third column shows the number of operations derived from the benchmarks on both the algorithms. The fourth and fifth columns show the MII and II calculated for both the algorithms. The sixth column shows the schedule density of FU (*with routing*). The seventh column shows the scheduling time in seconds for the mapping of the benchmark. The work in [Vassiliadis and Soudris, 2007b] uses the 8×8 CGRA array with 8 memory operations and Meshplus homogeneous architecture topology, row and column buses, predicate RF and data RF. MCHPSO was executed on an Intel Core 2 Duo CPU with 4 GB RAM and a clock speed of 2 GHz. Their algorithm was executed on a Pentium M 1.4 GHz PC. The comparison shows that MCHPSO was able to route the FFT benchmark with the minimal II, with a substantially smaller measure of execution time.

Table 3.12 shows the comparison of MCHPSO with the modulo scheduling algorithm used in [Dimitroulakos *et al.*, 2007]. Dimitroulakos et al., work uses a 2D CGRA with 16 PE with PEIT1 (*all PEs are connected with its row PEs and column PEs*) and PEIT2 (*nearest neighbor*) topology. The execution time is smaller in the PEIT1 than in PEIT2 because there is a smaller average routing delay experienced by PEIT2 which PEIT1 overcomes by the richer interconnection topology. The archi-

Table 3.11: Comparison of MCHPSO results with Mei et al work

Comparing algorithms		8 x 8 MCHPSO					Results reported in (Vassiliadis & Soudris, 2007)				
Benchmarks	*MII*	*# ops*	*II*	*Schedule Density (with routing)*	*OPC*	*Exe Time in Secs*	*# ops*	*II*	*Schedule Density (with routing)*	*OPC*	*Exe Time in Secs*
8X8 IDCT_hor	3	78	3	73.44	26	93.11	128	3	90.10%	42.7	340
4X4 FFT	3	67	3	75.52	24	105.9	79	4	75.00%	19.8	314

tecture has 2 scratch pad memories L0 and L1 and there are 2 memory buses per row in the 2D CGRA to fetch data from scratch pad memory L1 which quickly loads the data into the PE. The L0 scratch pad memory exploits this capability for reducing the memory accesses to L1 by reducing the data transfer bottleneck. That is achieved by storing the data reused values in the L0 and not fetching them again from the L1 memory. The topology used with our MCHPSO algorithm closely resembles the topology in PEIT1, described in Table 4 of [Dimitroulakos *et al.*, 2007]. Therefore, the work done in [Dimitroulakos *et al.*, 2007] based on PEIT1, was compared with the MCHPSO algorithm. The first column in Table 3.12 shows the benchmarks taken for comparison. The second and fifth columns show the number of operations in the benchmark. The third and sixth column show the II at which the algorithms were able to map the benchmarks. The fifth and ninth columns show the schedule density of FU (with routing) as calculated in Equation 3.2.

This comparative study has established that MCHPSO algorithm has a lower II for all benchmarks in spite of not using scratch pad memory, which has been used in [Dimitroulakos *et al.*, 2007]. The fifth benchmark 8×8 IDCT-hor depicts a typical case

of showing that the proposed algorithm maps at a lower II with the same number of operations and schedule density compared with results in [Dimitroulakos *et al.*, 2007].

The number of operations are different for the comparing algorithms because of the different analysis and transformation phase carried out in [Vassiliadis and Soudris, 2007a] and [Dimitroulakos *et al.*, 2007]. Not withstanding this discrepancy, the superior performance of the MCHPSO algorithm is evident. The MCHPSO algorithm finds schedules, with a minimal *II*, for all the benchmarks taken for comparison to the work done in [Vassiliadis and Soudris, 2007a] with a lower use of resources.

Table 3.12: Comparing MCHPSO with Dimitroulakos's et al work

Comparing algorithms		**4 X 4 MCHPSO**			**Results reported in (Dimitroulakos, Galanis, & Goutis, 2007)**		
Benchmarks	***MII***	***# of Ops***	***II***	***Schedule Density***	***# of Ops***	***II***	***Schedule Density***
latasynth	2	20	2	78.13	18	6	75
Volterra	4	28	4	57.81	27	7	70.3
IIR	4	36	4	78.13	39	8	59.5
4X4 FFT	5	67	7	81.25	95	17	69.6
8X8 IDCT_hor	6	78	7	89.29	79	14	85.1
latanal	2	20	2	68.75	18	8	62.5

3.4 Conclusion

In this chapter, we discussed the analysis of the Modulo Constrained Hybrid Particle Swarm Optimization (MCHPSO) algorithm for the loop scheduling problem in CGRAs. The results from MCHPSO algorithm indicate that the algorithm can find a valid schedule, placement and routing for the given benchmark loops, often with a minimal initiation interval, and with a low use of resources. To study the parallelizability of the MCHPSO algorithm, we have executed it on a quad-core machine with 8 logical processors and found good speedup. We also analyzed the MCHPSO algorithm with 2 different FU configurations. The experiment helped us to understand the enhancement in FU configuration increases the usage of FUs. Various interconnections in all FUs showed that increase in each additional edge produces a flexible routing process, thereby increasing the usage of resources. The size of RFs and the effect of topology have been studied to know the usage of registers and which topology worked the best for our scheduling problem. Shared RFs with each FU gave the lowest usage of registers. In the MCHPSO algorithm, the number of particles to be considered was studied and reported.

Chapter 4

Exploiting conditional structures onto CGRAs

4.0 Introduction

Coarse-grained reconfigurable architectures (CGRAs) have been structured for accelerating computation intensive parts like loops that require large amount of execution time. Loops, with conditional branches, have multiple execution paths which are difficult to perform software pipeline. In this chapter we review work done in handling conditional branches of loop, with if-then-else structures. We present an algorithm for scheduling predicated execution, with exclusivity feature, to exploit the conditional branches of loops. The performance of the proposed algorithm is compared with the predicated execution scheduling algorithm, with no exclusivity feature. The proposed algorithm finds a lower initiation interval for all the loops considered.

4.1 Background on HARPO/L

In this chapter we have taken DFGs generated from a HARPO/L program (*standing for HARdware Parallel Objects Language*). A HARPO/L program consists of a set of classes, interfaces, objects, and constants. The class declarations and interface declarations add new types to the type system, and the object declarations and constant declarations add objects to the object graph. The details of object declarations and constant declarations are similar to other object-oriented programming languages [Wu, 2011]. The grainless semantics of HARPO/L allows the object instantiation and connection to be done at compile-time, and at the run-time, there is no reference/pointer assignment.

The synthesized data flow graph (*DFG*) generated by the compiler [Wu, 2011] is very close to the representation of a schedulable datapath unit. All the benchmarks considered in this chapter are written as HARPO/L programs. In this chapter, we present limited details on HARPO/L and for more details please refer to [Wu, 2011].

4.2 DFG characteristics

This section describes the characteristics of the dataflow graph generated from the HARPO/L program. A DFG is a directed graph represented by a 5-tuple

$(N, E, type, I, O)$ where N is a set of nodes, E is a set of directed edges, $type$ is a function: $N \rightarrow NodeTypes$, I is a node representing the start of the graph, and O is a node representing the end of the graph [Wu, 2011]. Each node has an ordered set of input edges and an ordered set of output edges, and each edge has exactly 1

source node and exactly 1 target node. There are 2 kinds of directed edges between data flow graph nodes : $E = C \cup D$ where C is a set of control flow edges and D is a set of data flow edges. A data flow edge represents the synchronized transmission of a primitive value between data flow graph nodes. When a node is ready to receive data from an edge, it waits for the edge to be active, and once the edge is active, the node may receive the data and set the edge's activeness expires. When a node is ready to transmit data, it will transmit the data, set the edge active and wait until the edge is no longer active. The control flow edges are the edges transmitting only the activeness and no data. The symbols used for outgoing and incoming edges are $Z!$ means "activate control flow edge Z and wait until it expires and a control flow receive operation $A?$ means "wait until edge A is active, and set the activeness as expired" [Teifel and Manohar, 2004]. There are 13 types of data flow graph nodes. The graphic representations are shown in Figure 4.0. The description of the nodes used in the data flow graph are given below.

FETCH, VALUE and STORE: Each FETCH and STORE node are associated with a location. The operation fetch() means "fetch the value in the location". The operation store(a) means "store the value of a in the location". VALUE loads the data from the memory.

FUNC: It performs the assigned arithmetic or logic operation when all the incoming data edges are active.

COPY: Copies the incoming data to the various outgoing nodes.

START: The control flow of the whole graph starts from this node.

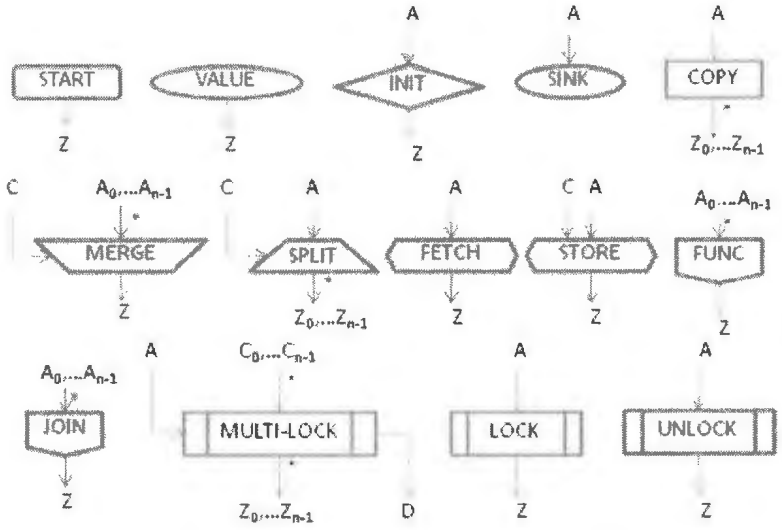


Figure 4.0: DFG node types, taken from [Wu, 2011]

JOIN: It joins all the control flow.

SPLIT and MERGE: SPLIT nodes are used to copy data to 2 different nodes based on the condition edge C. MERGE nodes take the result of the 2 execution paths based on the condition edge C.

MULTI-LOCK: each MULTI-LOCK node is associated with a number of locks, and indicates whether the locks are free. LOCK and UNLOCK node are associated with a lock, and indicates whether the lock is free.

4.3 Handling conditional statements

Loops, with conditional branches, have multiple execution paths and irregular flow of execution [Milicev and Jovanovic, 1998]. This seriously limits loops, with conditional branches, to exploit parallelism in CGRAs. The limitation in handling conditional

branches in CGRAs is that the configuration text cannot control the execution according to the computation results [Lee *et al.*, 2010]. The conditional branches part makes it hard to map the application onto CGRA, even though CGRAs can handle the most time consuming computation intensive part.

To tackle this problem, various solutions have been proposed in the literature. One of them is to perform predicated execution on the CGRA [Warter *et al.*, 1993]. In predicated execution, each processing element (PE) selectively executes an instruction according to its condition flag. This approach has the advantage of turning off unused PEs to reduce the power consumption. Predicated Execution restricts the parallel execution in CGRAs, because the condition should be checked before executing the statements inside the conditional statement [Smelyanskiy *et al.*, 2004], [eun Lee *et al.*, 2004].

The second approach is to run the application with speculation [Lee *et al.*, 2010]. Speculative Execution chooses one of the solutions depending on the condition, after executing all possible solutions first. This approach improves the performance, but consumes more power compared to the predicated execution.

The third approach is the Hierarchical Reduction, which collapses conditional constructs (e.g. if-then-else) into pseudo-operations. Next, list scheduling is employed on both the paths of the conditional construct and merging them into one path by taking the union of the resource usages along each path [Warter *et al.*, 1993]. Hierarchical Reduction does not assume special hardware support. Thus, after modulo scheduling, the code is regenerated by expanding the pseudo-operations. The fourth approach is called the Enhanced Modulo Scheduling [Warter *et al.*, 1992], which takes advantage of Predicated Execution and Hierarchical Reduction.

To support conditional branch in the reconfigurable architecture, the target architecture (TA) has to be modified slightly with an extended set of operations and additional ports [Chang and Choi, 2008], [Lee *et al.*, 2010]. Figure 4.1, shows the extension of arithmetic and logic unit (ALU) for predicated execution. The predicated instructions contain a condition flag to be executed first, which is supported as an additional port to the functional unit (FU). The difficulty that arises in mapping conditional branch on the CGRAs is to direct control flow to either stay in the current iteration path or to begin executing operations on a different iteration path.

In our target architecture, an additional port for each functional unit is added to support predicated operations. Predicated execution, with hardware support for conditional branching CGRAs, will be used in our modulo scheduling algorithm. To enhance the performance of predicated execution, we have developed an exclusivity feature algorithm, which will be discussed in the next section. We have implemented both the approaches of predicated execution without exclusivity and with exclusivity to study the performance of the exclusivity feature.

4.4 Predicated execution with exclusivity

4.4.0 Motivational example for exclusivity

Consider the DFG given in Figure 4.2, generated from the HARPO/L program having 1 if-then-else structure. Each node description is explained in Section 4.2 of this chapter. The node with number 250 in Figure 4.2, is a boolean node. There are 2 execution paths in the DFG, based on the boolean value of node 250.

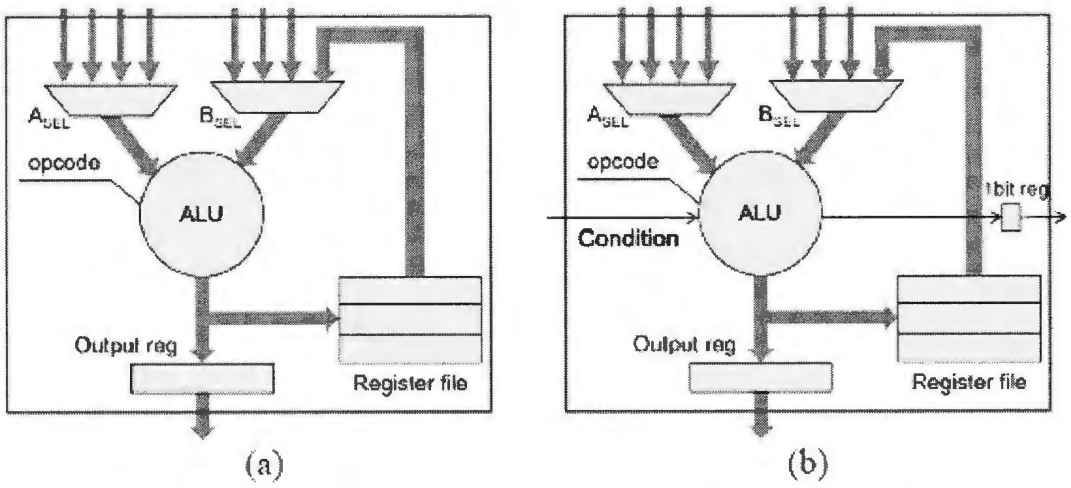


Figure 4.1: ALU modification for conditional branch a)original ALU b) modified ALU, taken from [Lee *et al.*, 2010]

Figure 4.3 compares the Modulo Reservation Table (*MRT*) of the 2 algorithms: predicated MCHPSO with exclusivity and predicated MCHPSO without exclusivity. Figure 4.3 displays only a few TA resources for comparison purpose. The predicates of the exclusive nodes in Figure 4.3 are given in Figure 4.4. In the MRT, functional unit F5 of cycle 0 has only node 500 with predicated MCHPSO no-exclusivity algorithm. In the predicated MCHPSO exclusivity algorithm, the nodes 500, 700, with predicates of 250 and $\neg 250$, are allocated. Since these nodes 500 and 700 are exclusive, i.e., both of these nodes will not be executed in the same iteration, 1 TA resource is enough. Using exclusivity only 4 register slots are used in resource R6 of cycle 1, with 5 DFG cells. Similarly 1 slot is used by 2 DFG cells in F5 and C01 of cycle 0. We can reduce the usage of TA resources by reusing the TA resources with exclusivity feature.

The exclusivity algorithm reuses the same resources that are exclusive with the current DFG cell, to be mapped in placement as well as in routing. Hence we propose

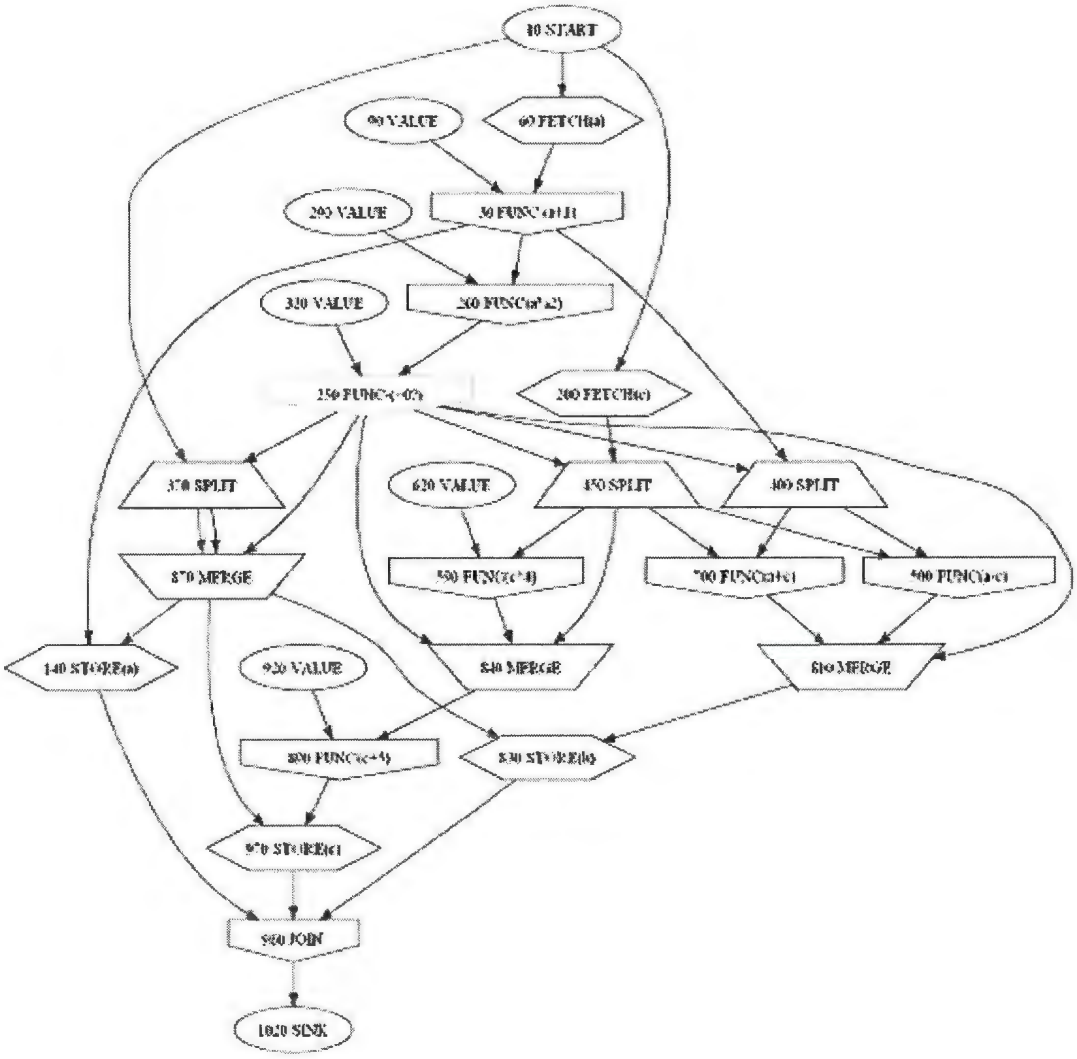


Figure 4.2: Example of HARPO/L DFG with if-then-else

Cycle 0			
	Ex	F5=[700, 500](1)	C01=[590-840, 450-840](1)
	NO-Ex	F5=[500](0)	C01=[](0)
Cycle 1			
	Ex	R6=[700-810, 250-840, 250-810, 30-400, 500-810](1)	
	NO-Ex	R6=[810-830, 250-810](0)	
Cycle 2	Ex	R3=[920-890, 320-250, 250-370, 250-450](0)	
	NO-Ex	R3=[290-260, 250-400, 250-370, 250-450](0)	

Figure 4.3: MRT Comparison of Exclusivity and No_Exclusivity Algorithm

DFG_Cells	Predicates
500	~250
700	250
590-840	~250
450-840	250
700-810	250
500-810	~250

Figure 4.4: Predicates of the exclusive nodes in Figure 4.3

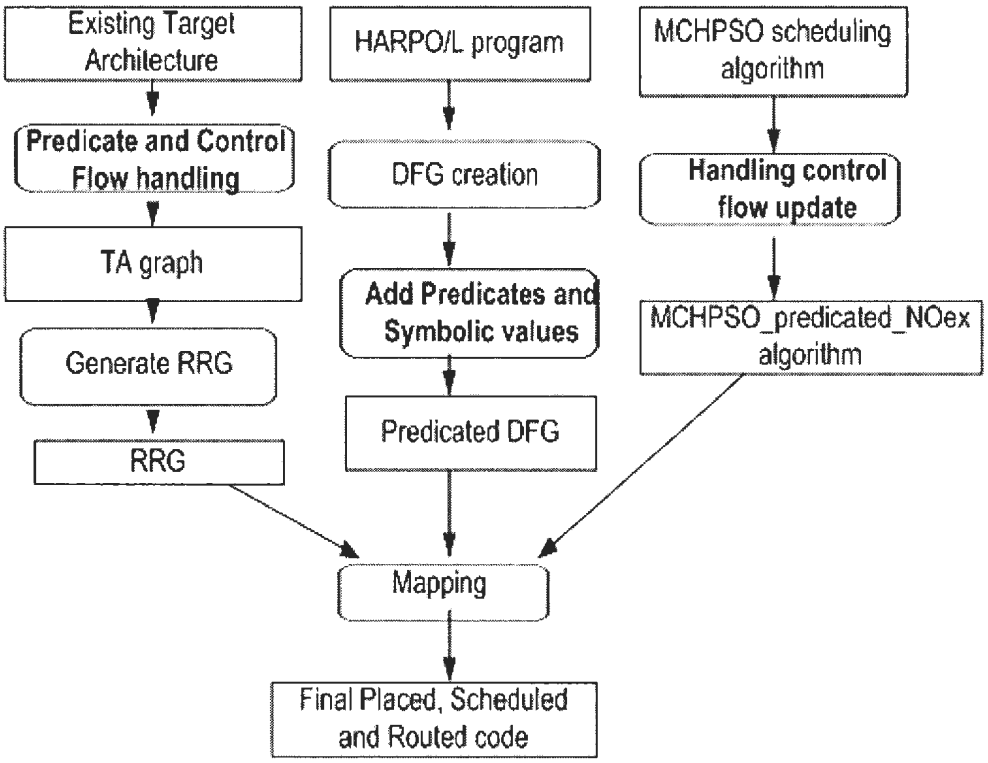


Figure 4.5: Predicated MCHPSO no exclusivity algorithm

the exclusivity feature, in addition to the predicated MCHPSO, to map if-then-else structure.

4.4.1 Mapping with MCHPSO predicated no exclusivity algorithm

4.4.1.0 Method description

The MCHPSO scheduling algorithm, discussed in the previous chapter, can place schedule and route DFG cells that have no predicates attached to them. There are 3 main updates needed to be done to the existing scheduling algorithm. We had to

update our TA graph, DFG and scheduler to handle execution paths and predicates to map if-then-else structure with our existing MCHPSO scheduling algorithm. The overall description of the predicated MCHPSO with no exclusivity algorithm is given in Figure 4.5. Each update is explained in the following subsections.

Adding predicates and symbolic values The main input to the scheduling algorithm is the data flow graph. In this chapter, we have generated DFG from the HARPO/L program, as described in early sections of this chapter. Each node in the DFG has a type. The condition nodes of an if-structure is assigned the node type *FUNC* and the outgoing edges of that node have an edge type *BOOL*. A condition node is shown in Figure 4.2, with the node number 250. From the condition node, the following nodes and edges in the DFG are controlled by the result of the BOOL value, having 2 execution paths of TRUE/FALSE. The outgoing edges of the condition node are assigned a symbolic value, based on the executed result of condition. Any node that has an incoming edge, with a symbolic value, assigns predicate to itself and to its successors with a combination of symbolic value and its predicate. The following subsection explains the assignment of symbolic values and predicates.

Assigning symbolic values When the DFG is created, all the DFG cells are assigned a TRUE value for their predicate and a null for their symbolic values. After all the nodes and edges are created, the symbolic values and predicates are added. Adding symbolic values is explained in Algorithm 4.0. First, the procedure starts to find all the condition nodes in the given DFG and adds them to the queue. Second, the procedure finds all the edges of the condition nodes and assigns the symbolic

```
Symbolicval_add(condition_list, DFG_cells)
Begin
    For each condition node (c1) in condition_list
        Add condition node to Queue
        c1_name:=c1.getName()
        Create a propositional variable (symval)
                                    with c1_name
        Assign c1's symbolic value as symval
        While Queue not empty
            Remove a node (n1) from Queue
            succ_symval:=symval
            For each successor(s1) of n1 from DFG_cells
                succ_type:=s1.getNode_type()
                n1s1:=Edge(n1, s1)
                Assign n1s1's symbolic value as succ_symval
                Assign s1's symbolic value as succ_symval
                if succ_type is COPY
                    Add s1 to Queue
            End For
        End While
    End For
End
```

Algorithm 4.0: Adding Symbolic values to DFG cells

value, based on the name of the condition node.

Assigning predicates Once the symbolic values are assigned to all the DFG cells, it is easy to assign the predicates. The adding of predicate values is explained in Algorithm 4.1. Each DFG cell is assigned a predicate value based on its parent cell. Mostly, all the DFG cells are assigned the same predicate of its parent cell. There are 3 special cases based on the node type. Figure 4.6 explains the 3 cases of node. The first case is a Condition Node (node 30), in which assigning a predicate to this node is always done with a AND operation with its existing predicate. The second case is a SPLIT Node (node 400). A SPLIT Node is assigned the same predicate of

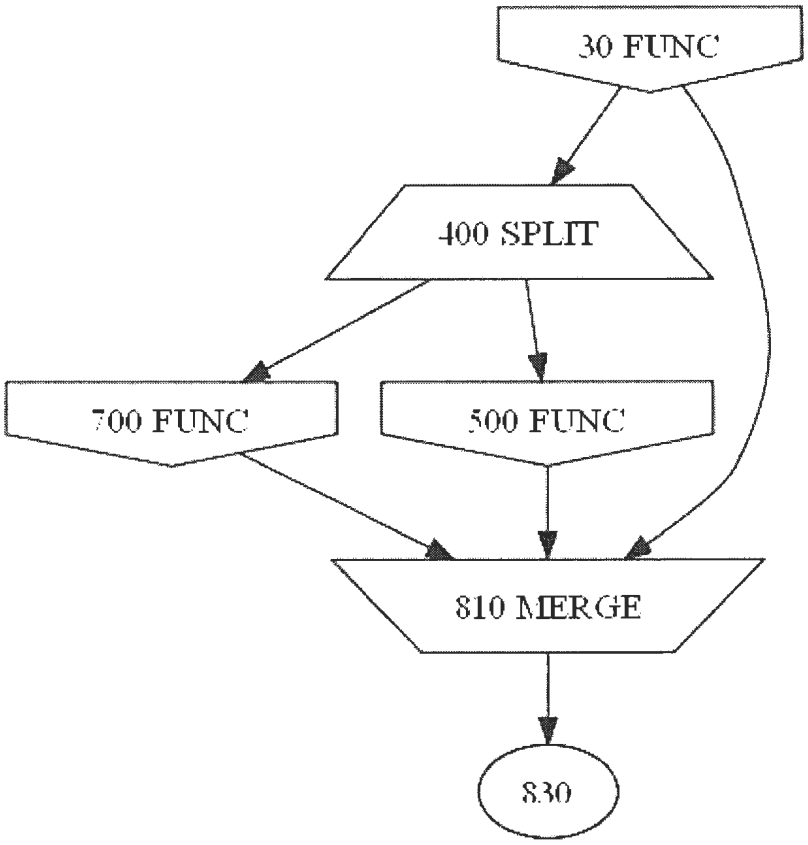


Figure 4.6: SPLIT and MERGE edges

its parent cell, but it predicts the predicate of its successor edge. A SPLIT Node has 2 edges, the first edge is for the TRUE value of condition and the second edge is for the FALSE value. The same predicate of SPLIT Node is assigned to the first edge. The negation of the SPLIT node's predicate is assigned to the second edge. The third case is a MERGE Node (node 810). The MERGE Node has 3 incoming edges, an edge with a symbolic value, an edge with TRUE predicate, and an edge with FALSE predicate. The MERGE Node is always assigned the same predicate value of the edge with symbolic value.

```
Predicates_add(condition_list, DFG_cells)
Begin
    For each condition node (c1) in condition_list
        Add condition node to Queue
        While Queue not empty
            Remove a node (n1) from Queue
            n1_predicate:=n1.getPredicate()
            n1_type:=n1.getNode_type()
            For each successor(s1) of n1 in DFG_cells
                FIRST_succ:=true
                succ_type:=s1.getNode_type()
                n1s1:=Edge(n1,s1)
                edge_type:=n1s1.edgeType(s1)
                //pass the same predicate if succ is not a SPLIT
                succ_pred:=n1_predicate
               if n1_type is SPLIT
                    if FIRST_succ
                        //FALSE execution path
                        Create a NOT node (not_n1) of n1_symval
                        Create an AND node (succ_pred) of n1_predicate
                                                          and not_n1
                        FIRST_succ:=false
                    else
                        //TRUE execution path
                        Create an AND node (succ_pred) of n1_predicate
                                                        and n1_symval
                        Assign n1s1's predicate as succ_pred
                    if succ_type is not MERGE or SINK
                        Assign s1's predicate as succ_pred
                        Add s1 to Queue
                    if edge_type is BOOL and succ_type is MERGE
                        //pass the predicate MERGE node's successor
                        Assign n1's successor predicate as succ_pred
                End For
        End While
    End For
End
```

Algorithm 4.1: Adding Predicates to DFG cells

TA predicate and control flow update The first update is done in the TA graph. We have to update the TA to accept conditions in the DFG and to handle the control flow in the DFG. Each functional unit in the TA was updated with an extra input port to handle predicated DFG cells. A DUMMY node was added in the TA to handle control flow in DFG cells. After the TA was updated, the functional units were ready to process predicates and control flow. Now the scheduler has to be updated to use the updated TA.

Handling control flow update in scheduling algorithm The inputs to the scheduler are the predicated DFG and the routing resource graph of the updated TA graph. The scheduler as presented in Chapter 3 can handle only the data flow in the DFG. To handle control nodes and control edges in the predicated DFG, the scheduler had to be updated. In the placement module of MCHPSO, the control nodes of the DFG are allocated to the DUMMY node of the TA graph. In the routing module with Dijkstra's algorithm, the control edges are not passed to check the resource availability to route. Instead control edges affect the schedule time of its successor nodes.

4.4.2 Mapping with MCHPSO predicated exclusivity algorithm

4.4.2.0 Method description

The MCHPSO predicated exclusivity algorithm has all the updates done in the previous section of MCHPSO predicated no exclusivity scheduling algorithm. There are 2

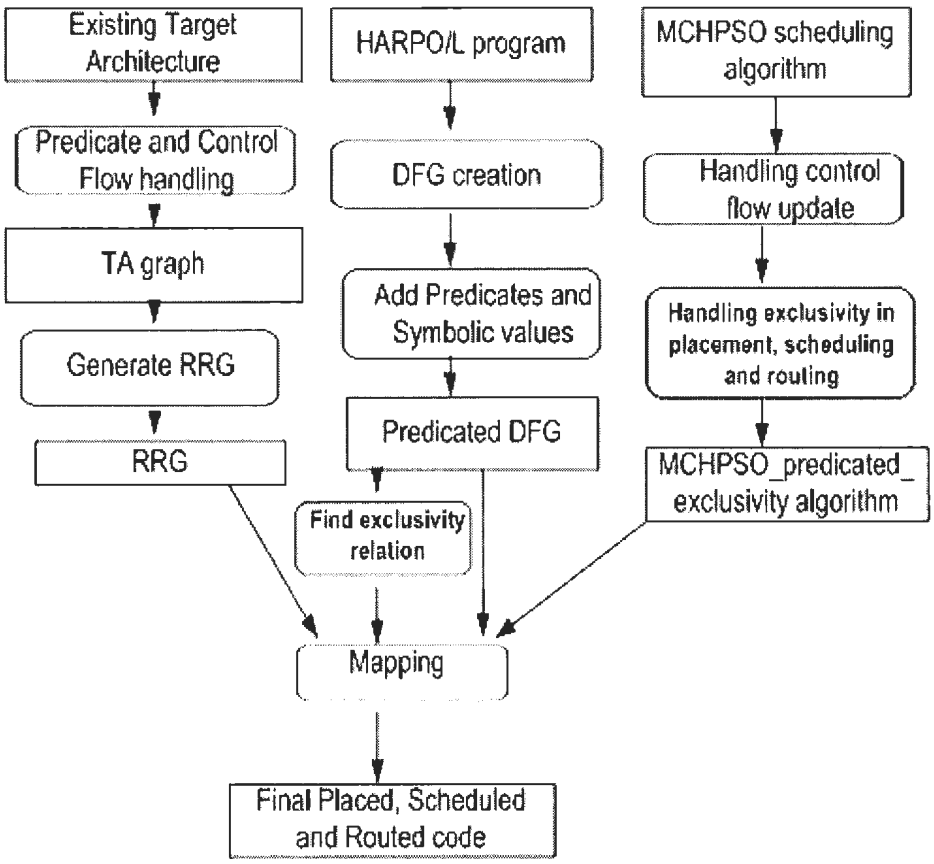


Figure 4.7: Predicated MCHPSO with exclusivity algorithm

extra updates done to the MCHPSO predicated no exclusivity scheduling algorithm to add the exclusivity feature. The first update is to find all the exclusive pairs of DFG cells. Second is to update our scheduler to handle exclusive DFG cells. The overall description of predicated MCHPSO with exclusivity algorithm is given in Figure 4.7. Each update is explained in the following subsections.

Find exclusivity relationship When there is a condition in a predicated DFG, we can find exclusive pair of cells which are on different execution paths. A cell can

```
Create_Exset(Alldfgcells)
For each cell_1 in Alldfgcells
    For each cell_2 in Alldfgcells such that cell_1 ≠ cell_2
        Create a boolean expression e with an AND node
            combining the predicates of cell_1 and cell_2
        if e is satisfiable
            Add the pair {cell_1, cell_2} to the exclusivity set
```

Algorithm 4.2: Creating exclusivity set

be either a node or an edge in the DFG. Two cells are exclusive, when both of these cells are on different execution paths i.e., both of these cells will not be executed in the same iteration. Algorithm 4.2 shows the steps to find exclusive pair of cells. Based on the predicates of each cell, all of its exclusive cells are found which are on different execution paths.

Handling exclusivity in placement, schedule and routing The scheduler update done in the predicated, no-exclusivity algorithm does not have methods to check for exclusivity in placement and routing. We have added an exclusivity check method both in the placement and the routing modules to place and route exclusive DFG cells. Each TA resource in the modulo reservation table has a set of DFG cells assigned to it during the execution of scheduling algorithm. In the MCHPSO predicated exclusivity algorithm, we propose to reduce the usage of TA resource and reuse existing resources based on exclusivity. Algorithm 4.3 shows the exclusivity calculation. Each DFG cell executes Algorithm 4.3 while searching for TA resource availability in placement and routing. The number of slots occupied in the existing DFG cells in the TA resource are found by the Maximum Independent Set (MIS) Algorithm 4.4. When a DFG cell wants to use a TA resource, the number of used slots in the TA resource must be less than the capacity of the resource. Adding the exclusivity algorithm to the predicated

exclusiveInSet(TAresource, dfgcells_existing, newOP)

1. If dfgcells_existing is empty add newOP and return true.
2. Else find MIS_SIZE= MIS(dfgcells_existing, newOP)
   a. Return MIS_SIZE≤Capacity(TAresource)
   b. Else return false.

Algorithm 4.3: Exclusivity check of TA resource

MIS(dfgcells_existing, newOP)

1. Create 2 empty sets MIS_search and MIS.
2. Add the dfgcells_existing and the newOP to the set MIS_search.
3. Find the degree of each cell in the set MIS_search based on exclusive pair.
4. Sort MIS_search set in ascending order of degree.
5. For each cell e1 in MIS_search
   If degree(e1)=0 then add e1 to the MIS set and remove e1 from MIS_search.
   else
   a. Check whether e1 is exclusive with the elements in the MIS set.
   b. If not exclusive add e1 to the MIS set.
   c. Remove e1 from MIS_search
6. Repeat step 5 until all MIS_search is empty.
7. Return the size of MIS set.

Algorithm 4.4: Maximum Independent Set of DFG cells

MCHPSO makes room for more DFG cells to be scheduled.

Algorithm 4.4 shows the steps to find maximum independent set of given cells.

4.5 Results

Modulo scheduling algorithms reported in the literature either jointly address inner loop mapping and predicated execution but do not consider CGRAs [Warter *et al.*, 1993], or consider modulo scheduling on CGRAs but cannot handle exclusivity i.e., can only address predicates and control flow, with no particle swarm optimization [Mei *et al.*, 2003b]. Thus, the novelty of our approach makes it difficult to experimentally

validate our results in comparison to previous work.

We have however devised an experiment that allowed us to assess the exclusivity feature for the complete modulo scheduling problem. Specifically, we compared the code generated by our exclusivity algorithm to code generated by a baseline algorithm that binds state of the art predicated code to a CGRA and then modulo schedules the code. In order to ensure fairness, the baseline algorithm (predicated no exclusivity algorithm discussed in Section 4.4.1) uses the same MCHPSO and target architecture implemented in our framework.

4.5.0 Experimental Set Up

The predicated MCHPSO with exclusivity check and without exclusivity check scheduling algorithm was written in Java and executed on an Intel Core 2 Duo CPU with 4 GB RAM and a clock speed of 2 GHz. To schedule an inner loop body with if-then-else structure requires 2 main inputs. The first input is the DFG generated from the HARPO/L programs with predication. The second input is the 4×4 and 4×3 TA graph. The 4×3 TA graph architecture is taken in this chapter to compare the performance of exclusivity with reduced TA resources. The predicated MCHPSO with exclusivity check and without exclusivity check algorithm places, schedules and routes the given DFG onto the TA. The modulo reservation table corresponding to the final schedule results is discussed in the next section.

Table 4.0: DFG characteristics of the benchmarks

Benchmark name	No of nodes	No of Edges	4 x 4 and 4 x 3 CGRA	
			MII	Sch_length
ifthen-1 condition	26	41	3	15
ifthen-2 conditions	52	87	6	18
ifthen-3 conditions	66	111	7	21

4.5.1 DFG characteristics

The characteristics of the DFG input to the scheduling algorithm are given in Table 4.0. The 3 benchmarks were written by me. The loop structure of the benchmarks are given in Figure 4.8, where s represents the statement and c represents the condition in the loop. The first column in Table 4.0 describes the benchmark name. The second and third columns list the total number of nodes and edges in the DFG to be mapped onto the TA. The fourth and fifth columns show the minimal initiation interval and schedule length.

4.5.2 TA characteristics

The TA graph has nodes and edges describing the details of the CGRA configuration. A detailed explanation of the TA is given in Chapter 3. Table 4.1 shows the resources available in a 4×4 and 4×3 CGRA. The first column shows the number of functional unit resources available. The second column shows the number of local registers

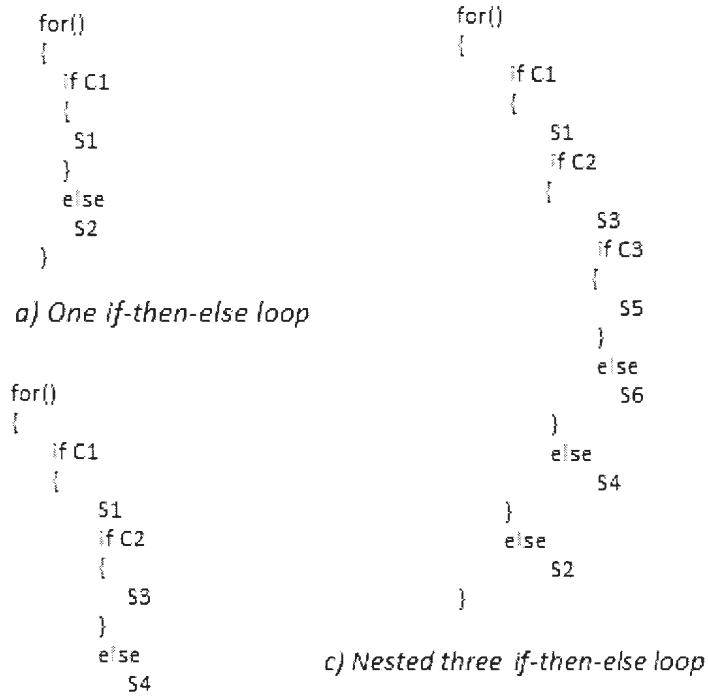


Figure 4.8: The first three benchmarks loop structure

available. The third column shows the number of shared registers available with memory loads and store. The fourth and fifth columns show the number of row and column buses available. The last column shows the number of total resources available. We have taken 4×3 CGRA to compare the advantage of predicated exclusivity in MCHPSO. The reduced number of resources in 4×3 CGRA makes it challenging for the predicated scheduling without exclusivity to place and route the DFG.

Table 4.1: Resources available in the Target Architecture

	#FU	#LRF	#SRF	#RB	#CB	# Total
4×4 –Target Architecture	16	48	4	4	4	76
4×3 –Target Architecture	12	32	4	3	4	55

4.5.3 Predicated Execution

4.5.3.0 With Exclusivity

The MCHPSO algorithm with predication execution and exclusivity feature was tested on 2 CGRA configurations. The exclusivity feature enables the TA resources to share the available MRT slots in routing as well as in placement. The sharing of resources reduces the total usage of MRT resources, making the remaining resources

available for other DFG operations. The following subsections describe the results obtained in the 2 CGRA configuration.

4×4 CGRA Table 4.2 displays the result obtained in a 4×4 CGRA, with predicated exclusivity algorithm. The first column shows the benchmark description. The second column shows the initiation interval (II) at which the algorithm was able to schedule the DFG. The third column shows the percentage of total functional unit usage in the MRT. The fourth column shows the percentage of total local register usage in the MRT. The fifth column shows the percentage of total shared registers usage in the MRT. The sixth and seventh columns show the percentage of total usage of column and row buses in the MRT. The eighth column shows the total resources available in the MRT. The ninth column shows the total resources used in the MRT. All the benchmarks were scheduled at the minimal initiation interval (MII) and minimal usage of resources. The total usage of the modulo reservation table of the final schedule was calculated by

$$\text{Res\_avail} = \sum_{\substack{\textit{Resource } r \\ \textit{All resource types in } TA}} (\#r \times Cap) \times II \tag{4.0}$$

$$\text{Res\_used} = \sum_{\substack{\textit{Resource } r \\ \textit{All resource in } TA}} (\#\text{slots\_used}) \tag{4.1}$$

$$\text{Usage}\% = (\text{Res\_used})/(\text{Res\_avail}) \times 100 \tag{4.2}$$

where,

Res_avail: Total TA resources in MRT

Res_used: Used TA resources in MRT

Table 4.2: Exclusivity results in 4 x 4 CGRA

Benchmark name	II	FU usage%	LRF usage %	SRF usage %	CB usage%	RB usage %	Available resources in MRT	Total resources used in MRT
ifthen-1 condition	3	52.08	29.86	83.33	33.33	16.66	228	84
ifthen-2 conditions	6	56.25	50.69	91.66	16.66	20.83	456	231
ifthen-3 conditions	7	64.28	59.52	92.85	17.85	21.42	532	309

Cap: Capacity of resource r.

#r: Total number of resources of type r.

#slots_used: Number of slots used in r

II: Initiation Interval

Ex_Usage%: Total Usage with Exclusivity algorithm

The same equations can also be used to calculate individual resource types.

4 × 3 CGRA Table 4.3 displays the result obtained in a 4 × 3 CGRA, with predicated exclusivity algorithm. The table fields description are same as explained for Table 4.2. Most of the benchmarks were scheduled at the MII with lower usage of total resources compared with predicated execution with no exclusivity algorithm. The resource usage is higher than the 4 × 4 CGRA utilizing most of the resources in 4 × 3 CGRA.

Table 4.3: Exclusivity results in 4 x 3 CGRA

Benchmark name	II	FU usage%	LRF usage %	SRF usage %	CB usage%	RB usage %	Available resources in MRT	Total resources used in MRT
ifthen-1 condition	3	69.44	44.79	83.33	41.66	33.33	165	86
ifthen-2 conditions	6	83.33	66.14	91.66	33.33	44.44	330	225
ifthen-3 conditions	8	88.54	80.07	81.25	34.38	50.00	440	339

4.5.3.1 No Exclusivity

The MCHPSO algorithm with no exclusivity feature in the predication execution was also tested on the 2 CGRA configurations. The MRT slots were not able to share the resources even when there was a critical need of resources in routing as well as in placement. Predicated execution with no exclusivity feature pushed the algorithm to its limit in some cases and couldn't find the schedule at lower II. The following subsections describe the results obtained in the 2 CGRA configuration.

4 × 4 CGRA Table 4.4 displays the result obtained in a 4 × 4 CGRA, with no exclusivity in predicated execution algorithm. The table fields description are the same as explained for Table 4.2. All the benchmarks were scheduled at the minimal initiation interval. The usage of total resources was higher when compared with predicated exclusivity algorithm.

Table 4.4: 4 x 4 CGRA results without exclusivity

Benchmark name	II	FU usage%	LRF usage %	SRF usage %	CB usage%	RB usage %	Available resources in MRT	Total resources used in MRT
ifthen-1 condition	3	70.83	35.41	83.33	8.33	33.33	228	100
ifthen-2 conditions	6	76.04	56.94	91.66	37.50	20.83	456	273
ifthen-3 conditions	7	87.50	76.19	92.85	57.14	42.85	532	408

4 × 3 CGRA Table 4.5 displays the result obtained in a 4×3 CGRA, with no exclusivity in predicated execution algorithm. The table fields description are the same as explained in Section 4.5.3.0 . Most of the benchmarks were scheduled at a higher II than the MII. The resource usage was higher than the 4×4 CGRA, with no exclusivity in predicated execution algorithm. When scheduling for the next higher II, the overuse of resources was reduced. The II and the resource usage was higher, when compared with predicated exclusivity algorithm.

4.6 Comparison

4.6.0 II achieved

Table 4.6 shows the initiation interval at which the final schedule was obtained. The final schedule result did not record overuse of resources and all DFG cells were scheduled, placed and routed. The II achieved in 4×4 CGRA configuration was the same

Table 4.5: 4 x 3 CGRA results without exclusivity

Benchmark name	II	FU usage%	LRF usage %	SRF usage %	CB usage%	RB usage %	Available resources in MRT	Total resources used in MRT
ifthen-1 condition	3	91.66	53.12	83.33	25.00	33.33	165	100
ifthen-2 conditions	7	89.28	74.10	85.71	42.85	47.61	385	287
ifthen-3 conditions	10	86.66	85.63	82.50	42.50	40.00	550	440

Table 4.6: II achieved in 4 x 3 CGRA and 4 x 4 CGRA

II achieved					
		4 x 4 CGRA		4 x 3 CGRA	
Benchmark name	MII	without exclusivity	Exclusivity	without exclusivity	Exclusivity
ifthen-1 condition	3	3	3	3	3
ifthen-2 conditions	6	6	6	7	6
ifthen-3 conditions	7	7	7	10	8

Table 4.7: Total usage of 4 x 4 CGRA

Usage % of total resources in MRT			
		4 x 4 CGRA	
Benchmark name	II	without exclusivity	with exclusivity
ifthen-1 condition	3	43.86	36.84
ifthen-2 conditions	6	59.87	50.66
ifthen-3 conditions	7	76.69	58.08

for both the scheduling algorithms with and without exclusivity feature. In 4×4 CGRA configuration, the II achieved was the same as the MII and thus both the algorithms achieved the best II. In 4×3 CGRA configuration, predicated exclusivity algorithm was able to achieve better result at lower II than the no exclusivity predicated algorithm.

4.6.1 Usage of resources in Exclusivity vs No exclusivity in 4×4 CGRA

Table 4.7 shows the usage of total resources in a 4×4 CGRA. Both the scheduling algorithms, with and without exclusivity feature, have found the schedule at the same II in 4×4 CGRA configuration. The final schedule of exclusivity predicated algorithm

Table 4.8: Total usage and overuse of 4 x 3 CGRA

Benchmark name	II	Usage % of total resources in 4 x 3 CGRA		Overuse % of total resources in 4 x 3 CGRA	
		NO-exclusivity	with exclusivity	NO-exclusivity	with exclusivity
ifthen-1 condition	3	60.61	52.12	0.00	0.00
ifthen-2 conditions	6	82.73	68.18	1.82	0.00
	7	74.55	61.56	0.00	0.00
ifthen-3 conditions	8	84.77	77.05	9.77	0.00
	9	82.42	71.52	5.25	0.00
	10	80.00	60.73	0.00	0.00

recorded lower usage of resources than the predicated execution with no exclusivity. Achieving lower usage of resources makes room in the CGRA to route more data and to use the available resources for executing more operations.

4.6.2 Overuse of resources in Exclusivity vs No exclusivity in 4×3 CGRA

Table 4.8 shows the usage and overuse of total resources in 4×3 CGRA. The overuse is the percentage of resource usage above 100 percent. In most of the benchmarks, exclusivity predicated algorithm found the schedule with lower II to be closer to the MII. The final schedule of exclusivity predicated algorithm recorded lower usage of

resources and lower II than the predicated execution with no exclusivity. The overuse of resources in predicated no exclusivity algorithm was caused by the unavailability of resources for placement and routing at the required time cycles. In case of exclusivity predicated algorithm, the overuse was avoided by sharing of exclusive resources. The exclusivity predicated algorithm made room for other DFG cells to be scheduled. The overuse of resources in the no exclusivity predicated algorithm decreased as II was incremented. Definitely exclusivity was able to save resources for future routing and placement even in smaller size CGRAs.

4.7 Conclusion

The objective of this chapter is to conduct a performance evaluation of exclusivity feature in the proposed MCHPSO algorithm with predicated execution. Under 2 different CGRA configurations, predicated MCHPSO with exclusivity was compared with predicated MCHPSO with no exclusivity feature. The proposed predicated exclusivity algorithm performance was very good even under lower resource availability.

A general conclusion from the result analysis, under 4×3 CGRA, predicated exclusivity algorithm was able to achieve scheduling with a lower initiation interval. While comparing the predicated exclusivity feature with predicated no exclusivity algorithm in 4×4 CGRA, the exclusivity enabled the scheduler to use fewer resources and provided more sharing of resources. The total usage of predicated exclusivity algorithm was lower than the predicated execution with no exclusivity.

The proposed exclusivity feature in predicated execution was experimented for if-then-else structures in the loops. It can also be extended to switch-case statements

and any condition-driven statements. The next chapter discusses the scheduling of nested loops onto the CGRAs.

Chapter 5

Recurrence exploitation in CGRAs

5.0 Introduction

A loop contains an inter-iteration dependence or recurrence if an operation in an iteration of the loop has a direct or indirect dependence upon the same operation from a previous iteration. To software pipeline a loop, a scheduler must handle inter-iteration dependencies, which arise from the loop's non-trivial recurrence circuits. In this chapter, the different approaches to solve the inter-iteration dependence in modulo scheduling are analyzed. By using a dynamic priority scheme, slack scheduling provides a novel integration of recurrence constraints and critical-path considerations. A priority scheme along with recurrence aware modulo scheduling is proposed to map inter-iteration dependencies onto Coarse Grained Reconfigurable Architectures (*CGRAs*). Our algorithm is aware of data dependencies caused by inter-iteration recurrence cycles.

5.1 Recurrence Handling

Recurrences form a cycle in the data-flow graph of the inner loop body. The scheduling slot of an operation depends on the schedule of the operands' producers, thus some operations in a recurrence cycle need to be scheduled before their producers have been placed. In a recurrent cycle, some operations are scheduled with only partial information of their producers' schedule [Oh *et al.*, 2009] affecting the overall performance of the loop schedule.

The II is constrained by the recurrences of the loop and by the resource constraints in the dependence graph. Inter-iteration dependences can induce recurrences that cause a maximum delay for the operations on the recurrence path or dependence cycle. Memory operations (load/store) are mostly the cause of a recurrence. These loop-carried dependences have a distance property, which is equal to the number of iterations separating the 2 instructions involved in the recurrence. If a dependence edge, $e(v, u)$, in a cycle has latency λ and connects the operations at $\delta_{v,u}$, then the recurrence constraint (*RecMII*), is calculated by

$$\text{RecMII} = \text{Max}_{c \in C} \left\lceil \frac{l}{d} \right\rceil \tag{5.0}$$

where,

- c is a recurrence cycle in the set of all cycles C of the DFG
  - l is the sum of all delay (λ) in the circuit
  - d is the sum of all distance $\delta_{v,u}$ in the circuit, variable $\delta_{v,u}$,denotes the distance between operation v and u, which means the operation u of iteration i depends on the operation v of iteration $i - \delta_{v,u}$.

The resource constraint ($ResMII$) is calculated from the resource usage requirements of the loop and it is derived from

$$ResMII = \text{Max}_{r \in R} \left\lceil \frac{\#r\ needed}{\#r\ available} \right\rceil \tag{5.1}$$

where,

r is a resource in the TA resources R.

Minimal Initiation Interval (MII) is a lower bound to start the pipeline scheduling process and it is computed as $MII = \max(RecMII, ResMII)$.

5.1.0 Motivational Example

The compilation flow with a motivational example is described in Figure 5.0. Consider the architecture configuration taken in Figure 5.0a, and a data flow graph (DFG) represented in Figure 5.0c. The architecture components in Figure 5.0a are functional units (FU) with a local Register File (RF). Figure 5.0b shows an routing resource graph (RRG) created by replicating the target architecture (TA) across 2 time cycles. The II is 3 for the DFG as it takes the maximum cycle delay from recurrences. The final embedding of DFG on RRG is shown in Figure 5.0 d.

As we are interested in mapping the recurrences (*i.e. inter iteration dependence*), we can see there is a loop carried edge from node *op* Z to node $c1$. The scheduling algorithm maps each operation to a FU and a time and maps each edge in the DFG to a path in the RRG. During the scheduling process, the algorithm keeps track of the resources being used in a modulo reservation table (MRT). The operation 2 is to be executed in $FU1$ at time 0 and therefore the $FU1$ is reserved for all cycles

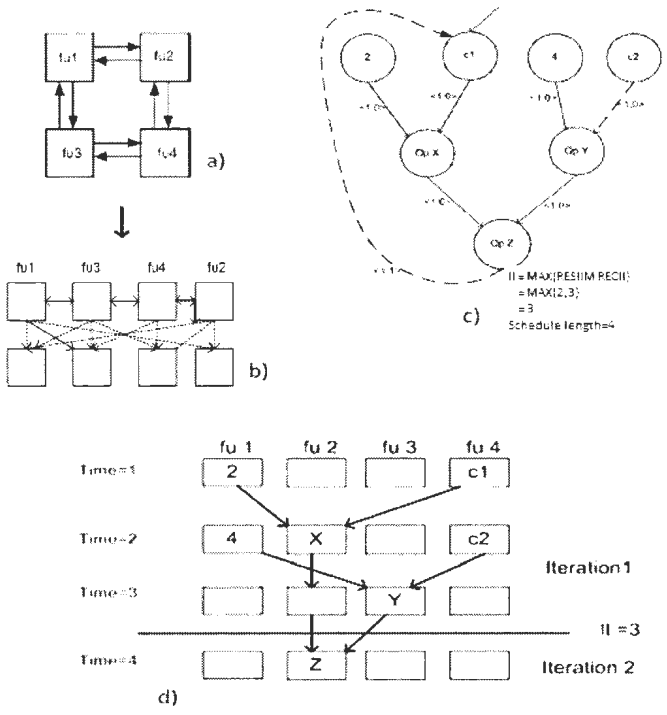


Figure 5.0: Motivating example a) 2 x 2 target architecture template instance, b) RRG, c) DFG and d) Final schedule, place and route

divisible by modulo II. Once a resource is reserved, it will not be available for the other operations in time cycles that have the same remainder modulo II. The routing path from operation 2 to operation X uses the RF of $FU1$, and a neighborhood connection from $FU1$ to $FU2$. The schedule given in Figure 5.0 d does not satisfy the recurrence constraints. The operation *op* Z has to be scheduled before operation $c1$ starts and this results in an invalid schedule. The scheduler did not take the priority to schedule the recurrence cycle before the other operations. To avoid maximum delay in the scheduling process and to efficiently schedule a recurrence cycle, an efficient modulo scheduler is needed. Hence a modulo scheduler is proposed, which gives priority to the recurrence cycles and finds a valid schedule in a short time. The proposed algorithm, with PSO and prioritized recurrence aware schedules, places and routes all the nodes and edges of a DFG onto the CGRA.

5.1.1 Existing Recurrence Handling Approaches

In this section, 4 different approaches have been used to modulo schedule the loops with inter-iteration dependencies are discussed. The four approaches that will be discussed for modulo scheduling are the rotation scheduling, bidirectional slack scheduling, edge-centric modulo scheduling (*EMS*), and recurrence-cycle-aware modulo scheduling (*RAMS*). The difficulty in handling loop carried dependence in CGRAs is that the quality of schedule depends on the partially scheduled operations and recurrences take long compilation time to find a valid schedule. The major reason for the degradation of the quality of a schedule in EMS is caused by speculative scheduling of operations that belong to a recurrence cycle [Park *et al.*, 2008]. In scheduling with simulated

annealing (*it is the DRESC method*) [Vassiliadis and Soudris, 2007b], a larger II is required to schedule the recurrence operations, which results in a very high execution time.

5.1.1.0 Rotation Scheduling

Rotation scheduling [Huff, 1993] takes various loop carried dependencies into consideration with its loop scheduling algorithm. In this approach, delays between loop carried dependencies are taken as constant or a function of the loop index. Rotation scheduling exposes parallelism across iterations with retiming. A retiming technique is used in rotation scheduling to rearrange registers to reduce the iteration period, that is to reduce the length of the critical path of the circuits.

Each rotation operation moves the schedule table of length L to length $L+1$ and finds a better intermediate schedule at the end of each rotation's iteration. A node remapping (*reschedule*) procedure is done at the end to reduce the static schedule. The final schedule is split into 3 parts:- rotation prologue (RP), a repetitive loop body (RB) and a rotation epilogue (RE). Rotation scheduling concentrates mainly on delays as a function and obtains an optimized schedule with an improvement in execution time.

5.1.1.1 Bidirectional Slack Scheduling

Bidirectional slack-scheduling method [Cho *et al.*, 2007] has been implemented in a FORTRAN compiler. This scheduler handles cyclic data dependencies, which arise from the loop's non-trivial recurrence circuits. Slack scheduling solves the recurrence problem by integrating recurrence constraints and critical-path considerations into an

operation-driven framework with limited backtracking.

The scheduler places operations one by one until either a feasible schedule is found or the heuristics give up. Slack scheduling can accommodate a novel bidirectional approach that attempts to schedule an operation either as early as possible or as late as possible, depending on a sophisticated heuristic. The heuristic's primary goal is to minimize each value's lifetime, in the hope that this will minimize the overall peak register pressure.

5.1.1.2 Edge-centric Modulo Scheduling

Edge-centric modulo scheduling (*EMS*) [Park *et al.*, 2008] schedules loops in an edge-centric way with a simple height-based scheduling priority scheme. In the EMS framework, the scheduling slot of an operation depends on the schedule of the operands' producers. The data-flow graph of a recurrence forms a cycle, thus some operations need to be scheduled before their producers have been scheduled. Consequently, some operations are scheduled with only partial information of their producers' schedule. First, the DFG of the target loop is converted into a reduced form by collapsing some nodes. The scheduling priorities of operations in the reduced DFG are calculated in such a way that simple edges get higher priority than high fan-out edges. When the scheduler places recurrence cycles, edges are placed even if their target operations are not yet placed. By calling the router function recursively for all operations in the cycle, the scheduler can put more effort into finding a legal mapping for the recurrence cycles.

5.1.1.3 Recurrence Aware Modulo Scheduling

The recurrence aware modulo scheduling (*RAMS*) [Oh *et al.*, 2009] scheme treats recurrence cycles in the DFG as a single unit. Instead of scheduling each operation individually, the algorithm first groups all operations in a recurrence cycle into a clustered node. The operations of a clustered node are then scheduled together. Clustering forms the recurrence cycles as a single node and transforms the DFG into an acyclic graph. Single nodes have priority during scheduling. The scheduler selects the clustered nodes according to their priority and schedules them one by one. All producers of the clustered recurrence cycle are now scheduled first even though some of them have a lower height than some operations in the recurrence cycle. After all clustered nodes have been scheduled, the remaining nodes are handled. A clustered node scheduling can be divided into 3 major steps: (1) scheduling of the incoming tree, (2) calculating the earliest scheduling time, and (3) scheduling the nodes of the clustered node. After all clustered nodes have been scheduled, the scheduler handles the remaining operations. The scheduler finds it more difficult to find a route for the remaining operations because most routing resources are already occupied.

RAMS prevents scheduling failures that arise due to redundant time constraints of operations that were scheduled before the recurrence cycles themselves. The whole process of scheduling is restarted if one of the recurrence cycles fails to be scheduled.

5.1.1.4 Comparison of Existing Approaches

The sparse interconnect and distributed register files in the CGRAs presents difficult challenges to a compiler to route the edges. Edge-centric modulo scheduling [Park *et*

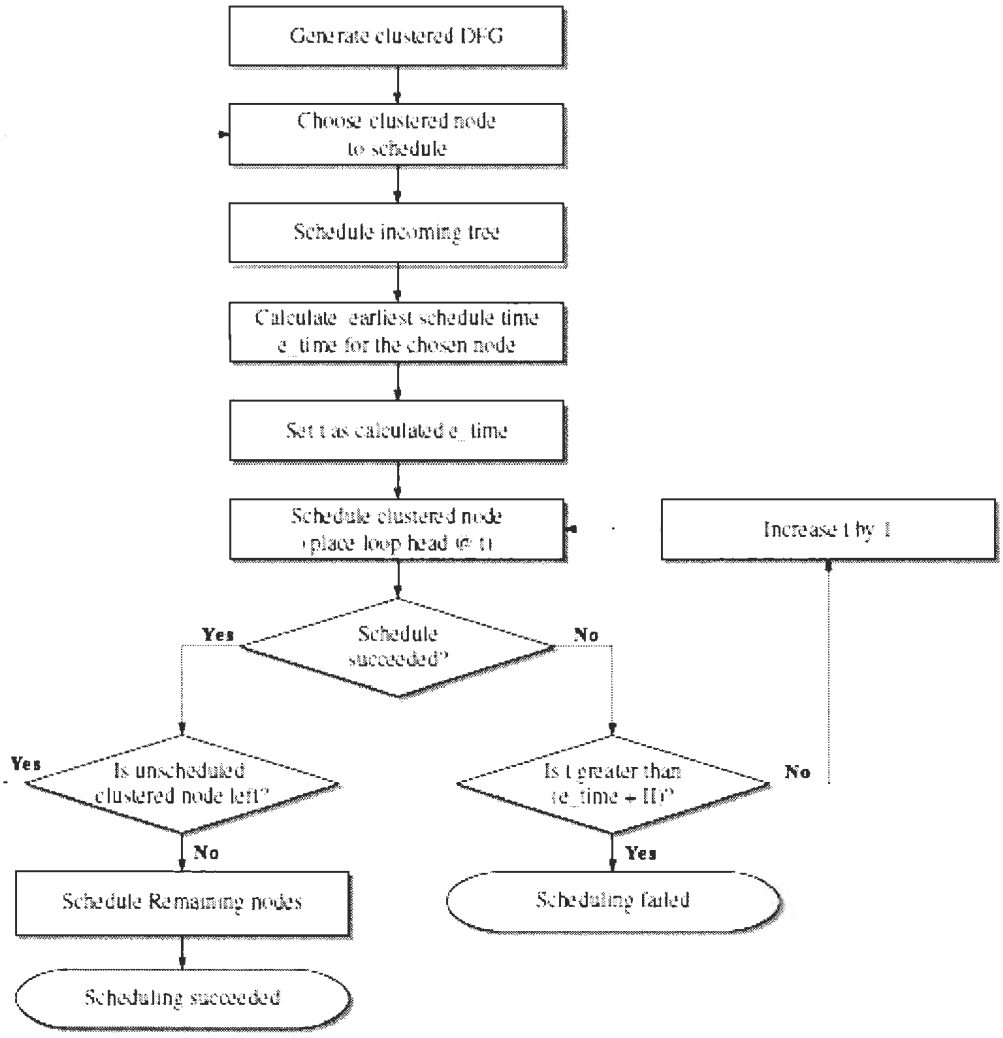


Figure 5.1: Flowchart of RAMS algorithm, taken from [Oh *et al.*, 2009]

al., 2008] concentrates on routing operands rather than node placement alone. Edges are categorized based on their characteristics, and the categories are used to route them during the scheduling process. Modulo scheduling with simulated annealing [Mei *et al.*, 2003a], takes longer compilation time but finds a better quality schedule. The EMS framework requires far less time than modulo scheduling with simulated annealing to find a schedule by sacrificing the quality of the schedule.

Recurrence aware modulo scheduling [Oh *et al.*, 2009] was able to achieve better quality schedules than the technique based on simulated annealing at a 170-fold speed increase. The scheduler in [Oh *et al.*, 2009] can only make decisions at the operation level of each edge. If the scheduler is not able to find a placement for the recurrence edge within II, the whole scheduling process repeats again with a larger II. In a dynamic priority scheme [Cho *et al.*, 2007], slack scheduling provides a novel integration of recurrence constraints and critical-path considerations. When the scheduler cannot find a slot for an operation, backtracking takes place by ejecting some operations. The bidirectional slack scheduler provides a lot of slack for the recurrence circuit to place them at the first place.

Considering all the difficulties of the above approach, a scheduler is needed that is fast enough to find a good quality schedule as well as give priority to the recurrence circuit.

```
RAP_PSO (DFG, TA)
begin
    II: = MII (DFG)
     Recurr_cycles:= Kosaraju(DFG)
    dfgList: = ComputeASAPandALAP (DFG)
     dfgList: = RecurrASLAP(recur_edges)
     sortedDFG: =Recurr_prioritySort (dfglist)
     max_schLength := findschLength(sortedDFG)
     schSucess := false
     trials :=0
     while !schSucess&& trials<NTRIALS do
          CreateRRG(TA, II, max_schLength)
          schSucess:=MCHPSO(sortedDFG, RRG, II, max_schLength)
          II++
          trials++
     end while
end
```

Algorithm 5.0: Mapping DFG with recurrences onto CGRAs

5.2 Proposed Method

5.2.0 Recurrence Aware Modulo Scheduling with Priority Scheme

In this thesis, a recurrence aware priority scheduler is proposed with a fast evolutionary, particle swarm optimization (*PSO*) called RAP_PSO. RAP_PSO is an extension of predicated exclusivity MCHPSO algorithm with added procedures supporting recurrence cycles in placement, routing and scheduling.

The recurrence cycles are modulo scheduled as early as possible when still relatively many resources are unoccupied. The priority scheme is applied to the DFG to give more priority to the nodes and edges of recurrence cycles. The overall procedure of the scheduling algorithm is shown in Algorithm 5.0.

Modulo scheduling starts with a minimal initiation interval (*MII*), as discussed in

Kosaraju(DFG)
var S : Stack[V]
topologicalSort$_{(V,E)}(S)$
{ all nodes are on S }
var $(V', E') :=$ transpose(V, E)
{ inv I (see below)}
    while S is not empty do
        val $u := S.top()$
        { u is in a terminal component of (V', E') }
        val $U :=$all nodes reachable from u in (V', E')
        { U is a terminal component of (V', E')}
        output U
remove each node in U from S and also from (V', E')
The invariant I is

- The nodes in V' are the same as the nodes in S.
- For any 2 nodes u and v in different components of (V', E'),
- if u is in a component that follows (in (V', E')) v's component, then in the original graph $u \longrightarrow v$ but $v' \longrightarrow u$, and so u is closer to the top of stack S than v.

Algorithm 5.1: Finding recurrence cycles with Kosaraju's strongly connected components algorithm

the background section. To schedule the operations of DFG, the ASAP (*As Soon As Possible*) time and ALAP(*As Late as Possible*) time are calculated for each operation in the DFG.

All the recurrence cycles in the DFG are found by the Kosaraju strongly connected component algorithm [Cormen *et al.*, 2009] described in Algorithm 5.1. If there is a path in the DFG from node u to node v and from v to u then u and v are said to be in the same strongly connected component. We write $u \rightarrow v$ to mean there is a path from u to v, i.e. it is reachable in 0 or more steps. In Algorithm 5.1 [0], all the nodes are first sorted topologically. Every time the top node u is popped off the sorted stack to find all nodes reachable from u in the transposed DFG, to form a strongly connected component. The list of nodes explored are a strongly connected component and are removed from the sorted stack. The above procedure repeats until all nodes in the sorted stack are explored. For detailed explanation with examples, please refer to [Cormen *et al.*, 2009].

Once the recurrence cycles are found, ASAP and ALAP times are calculated. For all the nodes, ASAP and ALAP are calculated by ignoring the back edges or loop-carried edge ($opZ \rightarrow c1$), as shown in Figure 5.0 as dotted line. The node opZ is called the loop head or source (LH) and the node $c1$ is called the loop tail or target (LT). The source node of the loop carried edge's ALAP is updated to limit its mobility with the target node as shown in

$$\text{source\_ALAP} = \text{target\_ALAP} + (\text{distance} \times II) - delay(source) \tag{5.2}$$

[0]The given simplified version of the algorithm was written by Dr. Theodore Norvell.

where,

- distance is the iteration difference between source and target node
- delay is the processing time of the node

As these loop carried edges are modulo constrained, they are affected by the II value in the scheduling time. Once all the nodes in the DFG are assigned the correct earliest and latest times, the RAP_PSO scheduler starts with the recurrence aware prioritized DFG and the RRG generated from the TA graph. In the recurrence aware prioritized DFG, all the recurrence cycles are given higher priority than the remaining operations. The routing procedure of Dijikstra's algorithm checks every recurrence edge satisfying the equation

$$\text{source\_schtime} + \mathit{delay}(\mathit{source}) \leq \text{target\_schtime} + \text{distance} \times II \tag{5.3}$$

for the placed and scheduled particles in RAP_PSO scheduling algorithm. If the particles do not satisfy the Equation 5.3, next generation of particles continue to explore a valid quality schedule.

The RAP_PSO scheduler takes each particle to find a valid schedule, placement and routing for all the operations and edges in the DFG. The particles are initialized with random schedule time and placement. Next the scheduler finds the routing resources for the edges and gives priority to loop carried edges. The routing results with the number of edges routable and routing cost, are taken as the fitness value for the particles. Once a final schedule is obtained, the scheduler checks whether all the nodes and edges being mapped satisfy the resource constraints, recurrence constraints, schedule time validity (Figure 5.2) and modulo constraint. If the schedule is not valid,

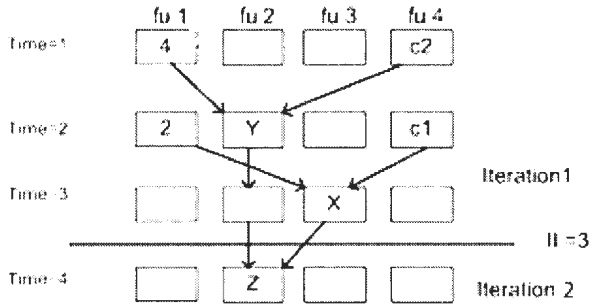


Figure 5.2: Successful final schedule for the DFG shown in Figure 5.0

the II is incremented by 1 and the scheduling process is repeated. The final correct schedule for the DFG shown in Figure 5.0, is given in Figure 5.2. The schedule satisfies modulo constraint, resource constraints, recurrence constraints and schedule time validity.

5.2.1 Architecture Extensions to Speedup Recurrence Handling

In the existing target architecture, the memory load and stores of operands (*called live-in/live-out*) were initially available in the shared register file. The top row of functional units (*FUs*) were mainly used for Memory Unit (*MU*) operations. These FUs were rarely used by other operations and it decreased the bandwidth to move the live-in operands to later cycles. To increase the bandwidth, an extension has been adapted as suggested in [Oh *et al.*, 2009] to add a dedicated register file (*RF*) to each read port of the RF that contains the live values.

Dedicated RFs do not suffer from critical path delay because it takes 1 additional

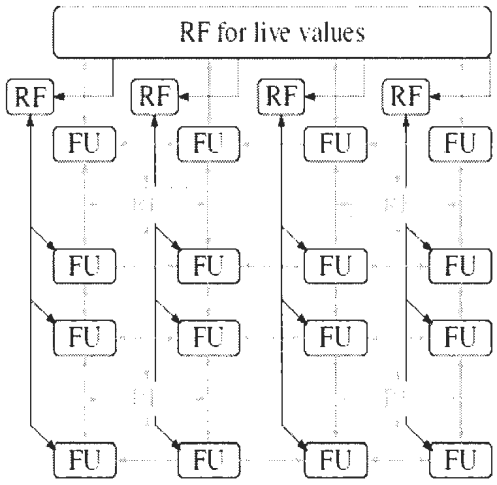


Figure 5.3: CGRA architecture with dedicated RFs for live values, taken from [Oh *et al.*, 2009]

cycle to access a live value through a dedicated RF [Oh *et al.*, 2009]. The same live-in values can be retained for several cycles in dedicated RFs and it increases the output bandwidth. Since all FUs now have indirect access to the live-in values, the dedicated RF reduces the number of resources used for routing live-in values. The results of using this extended architecture and its performance are discussed in the next section.

5.3 Discussion of Results

5.3.0 Experiment Set Up

The RAP_PSO with recurrence aware scheduling algorithm was written in Java and executed on an Intel Core 2 Duo CPU with 4 GB RAM and a clock speed of 2 GHz. To schedule an inner loop body with loop carried edges requires two main inputs. The

first input is the prioritized DFG with recurrence cycles identified from the HARPO/L programs. The second input is the 4×4 or 4×3 improved target architecture graph with extensions. The RAP_PSO algorithm places, schedules and routes the given DFG onto the TA by correctly mapping the recurrence edges. The usage of target architecture is found from the Modulo Reservation Table and is discussed in the next section.

5.3.1 DFG with Recurrences

The characteristics of the DFG input to the RAP_PSO scheduling algorithm are given in Table 5.0. The livermore loops benchmarks were taken from [Peters and Square, 2011] which are written in language C. The first column describes the benchmark name. The benchmarks were selected such that they have recurrence cycles in them for scheduling. The benchmarks were rewritten in HARPO/L language for the inhouse compiler to generate data flow graphs. The data flow graph generated from the compiler goes through a preprocessing and analysis stage for scheduling. In the preprocessing and analysis stage, the DFG is optimized with variable usage and the inner loop body is retrieved with recurrence edges in them. The second and third column list the total number of nodes and edges in the DFG to be mapped onto the TA. The fourth and fifth columns show the minimal initiation interval and the schedule length.

Table 5.0: Recurrence Benchmark Characteristics

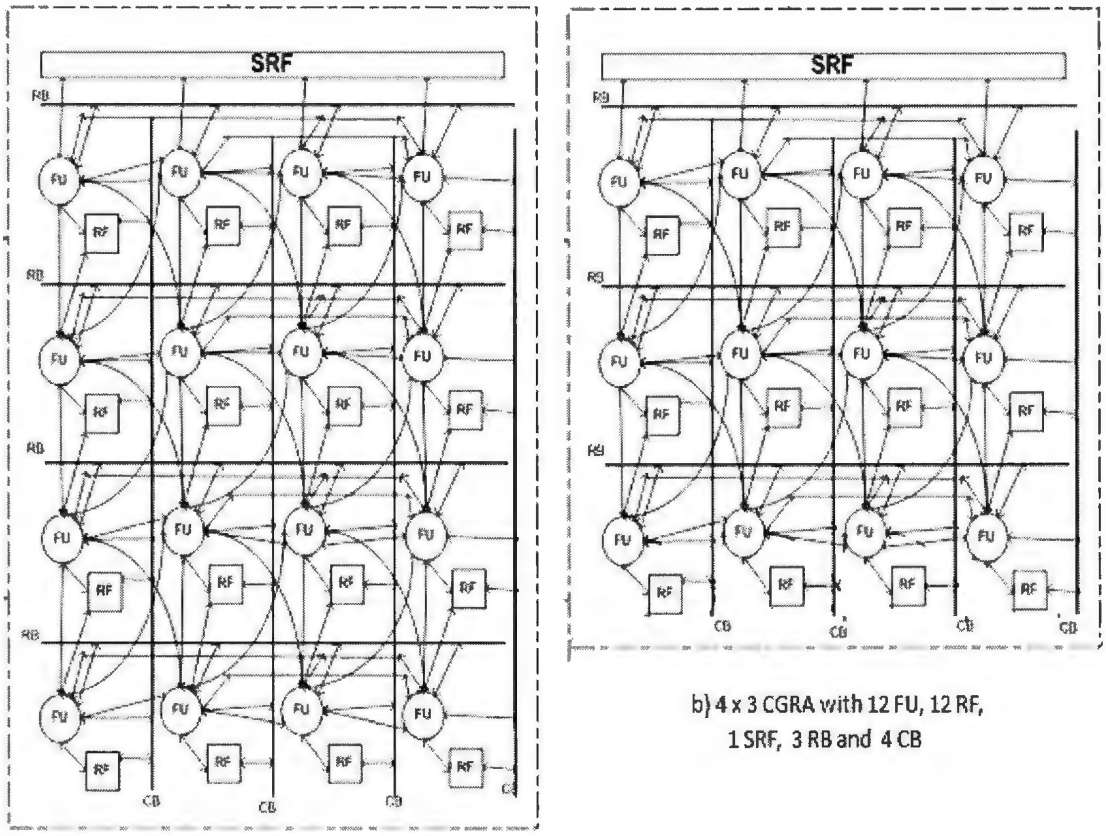
Benchmark name	No of nodes	No of Edges	ResMII	RecMII	MII	Sch_length
Livermore_ recurreqn1	11	18	2	3	3	9
Livermore_ condrecurr	28	50	4	4	4	16
Livermore_ matrixmul	34	49	4	4	4	16
Livermore_ tridiagonal	8	13	2	2	2	8
Livermore_ recurreqn2	10	15	2	2	2	8

5.3.2 TA Characteristics

The TA graph has nodes and edges describing the details of the CGRA configuration. Figure 5.4 shows the resources available in a 4×4 and 4×3 CGRA. The reduced number of resources in 4×3 CGRA makes it challenging for the routing of recurrence cycles and other data dependencies in the DFG.

5.3.3 4×4 CGRA recurrence schedule results

Table 5.1 displays the result obtained in a 4×4 CGRA, with RAP_PSO scheduling algorithm. The first column shows the benchmark description. The second column shows the initiation interval at which the algorithm was able to successfully schedule the DFG. The third column shows the percentage of total functional unit usage in the MRT. The fourth column shows the percentage of total local register usage in the



a) 4 x 4 CGRA with 16 FU, 16 RF, 1 SRF, 4 RB and CB

b) 4 x 3 CGRA with 12 FU, 12 RF, 1 SRF, 3 RB and 4 CB

Figure 5.4: Comparison of 4 x 4 and 4 x 3 architecture configurations

Table 5.1: Recurrence schedule results in 4 x 4 CGRA

4 X 4 CGRA- MCHPSO_RAP								
Benchmark name	II	FU usage%	LRF usage %	SRF usage %	CB usage%	RB usage %	Available resources in MRT	Total resources used in MRT
Livermore_ recurreqn1	3	22.92	11.80	41.66	8.33	8.33	228	35
Livermore_ condrecurr	4	64.06	28.13	81.25	31.25	18.75	304	116
Livermore_ matrixmul	4	75.00	43.22	81.25	25.00	12.50	304	150
Livermore_ tridiagonal	2	21.88	7.29	50.00	0.00	37.50	152	21
Livermore_ recurreqn2	2	28.13	9.38	62.50	50.00	0.00	152	27

MRT. The fifth column shows the percentage of total shared registers usage in the MRT. The sixth and seventh columns show the percentage of total usage of column and row buses in the MRT. The eighth column shows the total resources available in the MRT. The ninth column shows the total resources used in the MRT. All the benchmarks were scheduled at the MII and with minimal usage of resources. The total usage of the modulo reservation table and individual resource usage of the final schedule are calculated as in Chapter 4.

Table 5.2: Recurrence schedule results in 4 x 3 CGRA

4 X 3 CGRA- MCHPSO_ RAP								
Benchmark name	II	FU usage%	LRF usage %	SRF usage %	CB usage%	RB usage %	Available resources in MRT	Total resources used in MRT
Livermore_ recurreqn1	3	27.77	17.70	41.66	8.33	11.11	165	34
Livermore_ condrecurr	4	83.33	40.63	81.25	58.33	40.00	220	118
Livermore_ matrixmul	4	87.50	68.75	81.25	37.50	41.66	220	154
Livermore_ tridiagonal	2	29.16	10.93	50.00	12.50	33.33	110	21
Livermore_ recurreqn2	2	54.16	17.18	62.50	12.50	0.00	110	30

5.3.4 4×3 CGRA recurrence schedule results

Table 5.2 displays the result obtained in a 4×3 CGRA, with RAP_PSO scheduling algorithm. The first column shows the benchmark description. The second column shows the initiation interval at which the algorithm was able to schedule the DFG with resource and recurrence constraints. The third column shows the percentage of total functional unit usage in the MRT. The fourth column shows the percentage of total local register usage in the MRT. The fifth column shows the percentage of total shared registers usage in the MRT. The sixth and seventh columns show the percentage of total usage of column and row buses in the MRT. The eighth column shows the total resources available in the MRT. The ninth column shows the total resources used in the MRT. All the benchmarks were scheduled at the MII with lower usage of resources. The total usage of the modulo reservation table and individual resource usage of the final schedule are calculated as in Chapter 4.

Comparing the results of 4×4 CGRA and 4×3 CGRA we find that both were able to schedule at MII. The resource usage in 4×3 CGRA was higher than 4×4 CGRA. Most of the critical resources are used in 4×3 CGRA and was able to route within MII. The functional units usage was higher by 4.85% to 26.04%. The local register files usage was higher by 3.64% to 25.53%. The row bus usage was higher by 2.78% to 29.16%.

5.4 Conclusion

In this chapter, four approaches to solve the loop scheduling problem with recurrence were discussed. The schedule results of both edge-centric schedulers, EMS and RAMS, outperform DRESC [Mei *et al.*, 2002] by two orders of magnitude. While the RAMS is about 2 times slower than EMS, the superior scheduling quality of RAMS over EMS compensates for this slowdown. An algorithm is proposed based on RAMS and dynamic priority to solve the loop scheduling problem with loop carried dependencies. The proposed algorithm takes the advantage of PSO to speed up the scheduling process combined with recurrence aware priority to obtain a good quality schedule. The proposed RAP_PSO algorithm was tried on the livermore loops benchmarks. The recurrence cycles found in the benchmarks was modulo scheduled at minimal II.

Chapter 6

Conclusions and Future Work

6.0 Contributions

Today's embedded systems such as 4G mobile phones, tablet computers or personal digital assistants (*PDA*) requires very high computing speed in multitasking applications, downloading of video streams and to handle the high-speed wireless data communication. Coarse-grained reconfigurable architectures (*CGRAs*) are emerging as potential solutions for the above challenges. CGRAs bring advantages such as high performance, low communication overhead, high flexibility and ease of programming. In this thesis, a CGRA is taken to address the problem of mapping application with loops which consume lot of computation resources.

Applications such as multimedia and telecommunication systems with audio, video encoders and digital signal processing consume a long time in compilation with the presence of repeating loop statements. In this thesis, we have considered the problem of scheduling, placing, and routing loops for CGRAs. The mapping problem for

coarse-grained reconfigurable architectures is NP-hard in general. Software pipelining the loops, requires an efficient modulo scheduling algorithm. A modulo constrained hybrid particle swarm optimization algorithm (*MCHPSO*) [Gnanaolivu *et al.*, 2010a] is proposed for scheduling critical loops.

MCHPSO combines the features of the evolutionary approach of PSO and a mutation operator to find potential solutions for the modulo scheduling problem. A particle in the PSO system finds a placement for the operations in the loop body, a scheduling time at which an operation can be executed and a routing path for the operands. The solution search was challenged by the critical resources available in the CGRAs, modulo constraints to reserve the resources for repeated iteration, and the complexity of the loop. MCHPSO managed to schedule most of the loops in the minimal initiation interval while taking very little execution time. The proposed algorithm was successfully tested on 8 standard benchmarks from digital signal processing (*DSP*) applications.

In the experimental demonstration of MCHPSO [Gnanaolivu *et al.*, 2011a], it was found that a parallel search with 10 particles was enough to find a valid solution. MCHPSO was able to avoid local optima by exploring and exploiting more solutions than the DRESC [Mei *et al.*, 2002] in the time-space graph of the target architecture. It was also discovered that MCHPSO was able to increase its scheduling speed when the interconnections between the functional units (*FUs*) are more flexible. The MCHPSO was able to efficiently use shared registers in a shared register file (*RF*) interconnection architecture template. MCHPSO speedup was analyzed by executing the algorithm in an Intel core i7 machine. The proposed algorithm was able to parallelize the search for a scheduling solution in the 8 logical threads present in the

i7 machine, and achieve good speedup. The proposed algorithm achieves better resource usage with lower initiation interval and efficiently maps with a minimal time compared to DRESC [Mei *et al.*, 2002].

Various configurations of the ADRES template were tried with the MCHPSO algorithm. Out of these, results corresponding to 8×8, 4×4 and 4×3 CGRAs are reported in the thesis. The most interesting challenge was to schedule conditional loops [Gnanaolivu *et al.*, 2010b] with if-else statements on the 4×3 CGRA configuration. MCHPSO with predicated exclusivity feature handled the challenge to place as well as route with lowest possible initiation interval (II). The 4×3 CGRA configuration performed as well with resource utilization as the 4×4 configuration.

The minimal initiation interval to repeat the modulo schedule of an iteration depends on resource constraints as well as recurrence constraints. Loop carried dependencies were mapped on the CGRA with a recurrence aware priority scheme applied to MCHPSO called RAP_PSO. The proposed RAP_PSO algorithm [Gnanaolivu *et al.*, 2011b] was tried on 5 recurrence benchmarks from the livermore loops. The proposed algorithm scheduled efficiently on the 4×3 CGRA configuration.

The proposed MCHPSO with exclusivity feature and recurrence aware scheme was able to place, schedule and route the inner loop body of a critical application.

6.1 Suggested Future Work

The proposed modulo constrained hybrid particle swarm optimization algorithm with exclusivity feature and recurrence awareness worked well on the benchmarks considered. MCHPSO algorithm was able to map application loops written in the C lan-

guage and HARPO/L. Many opportunities exist to perform further research around this work. More experiments can be done to evaluate the proposed algorithm on complex benchmarks which include nested loops, switch-case statements, pointers and so on. To find the suitability and effectiveness of the proposed algorithm, it could be compared with various other modulo scheduling algorithms and heuristic methods such as iterative modulo scheduling [Rau, 1994], DRESC [Vassiliadis and Soudris, 2007b], recurrence cycle aware modulo scheduling [Oh *et al.*, 2009], clustered modulo scheduling [Sánchez and González, 2001], swing modulo scheduling [Llosa *et al.*, 1996], hypernode reduction modulo scheduling [Llosa *et al.*, 1995], modulo scheduling with integrated register spilling [Zalamea *et al.*, 2001]. RAP_PSO algorithm can be tried on several coarse-grained architectures to compare with existing approaches RAMS and DRESC.

The current work can be extended to place, schedule and route an entire application with many loops and non loop statements. The target architecture configuration can be extended to handle non loop statements and loop statements. The work could also be extended to exploit task-level parallelism (*TLP*) as well as instruction-level parallelism (*ILP*) and loop-level parallelism (*LLP*). Modulo scheduling a complex application presents a big challenge even to existing architectures such as ADRES [Mei *et al.*, 2005b] due to its computational complexity. Modulo scheduling experiments conducted on a H.264/AVC decoder by Mei et al. [Mei *et al.*, 2005b] shows that ADRES architecture and its compiler provide many features that are critical for mapping a complex application. Hence with MCHPSO it is possible to map complex applications imposing a performance and power usage challenge.

In order to improve the existing MCHPSO algorithm, the following enhancements

are suggested. MCHPSO can be improved to find even lower initiation intervals by improving the bandwidth. The resource initiation interval is normally affected by the memory units available for the live-in values.

In mapping loops onto CGRAs, few algorithms have been tried with the evolutionary approach. Many efficient algorithmic approaches like genetic algorithms, ant colony algorithms or hybrid combination of evolutionary operators can be tested for the modulo scheduling problem and compared against the proposed MCHPSO algorithm. The preprocessing stages for the data flow graph can be extended to handle complex control structures and to select which portions of an application will be executed on the CGRA and which will be executed on a microprocessor.

There are number of open issues in the CGRAs that can be solved such as self-reconfiguration, power efficient design of memory ports and data streaming, checking graph isomorphism for complex graphs, studying the strength of functional units, and system flexibility.

6.2 Concluding Remarks

Reconfigurable computers compute a function by configuring functional units and they are able to achieve high speed, low energy consumption and low power requirements. Reconfigurable computing systems are upgradeable and can serve as an affordable, fast, and accurate tool for verifying electronic designs. Coarse-grained reconfigurable architectures are efficient for long running computations, DSP, video and image processing.

Compiling applications for CGRAs usually involves the following tasks: dataflow

analysis and optimization of the application, creation of a target architecture graph, and the scheduling algorithm. The scheduler in the compilation process involves 3 tasks: scheduling, placement, and routing. Scheduling assigns the time cycle to execute the operation, placement assigns a functional unit and routing takes care of moving data from producer functional unit to consumer functional unit. An effective compilation mainly depends on the scheduler handling all the constraints on both the application and the architecture. In this thesis, a new scheduling algorithm is proposed with an evolutionary approach.

Evolutionary algorithms are best employed when there is no feasible optimization approach. In the modulo scheduling problem, the evolutionary approach is used to determine an optimized solution in resource usage and efficient mapping. Particle swarm optimization is primarily suited for numerical optimization problems. To avoid local optimal solutions, PSO with a heuristic operator is employed to solve the modulo scheduling problem. The implementation of MCHPSO was very successful in solving the modulo scheduling problem with optimal or near optimal initiation interval and low usage of resource with no overuse.

Mapping loops onto reconfigurable architectures still leaves many challenges open. For example, in our current work we assumed that loop iterations execute in the pipeline to develop a mapping flow that works reasonably for many applications. However, some loops might be better mapped when iterations execute in parallel. Therefore, the mapping style could be another dimension for optimizing the mapping of different applications. Furthermore, the current mapping flow has several constraints on architectures and application loops that must be relaxed. Our future research will investigate mapping techniques for more different classes of reconfig-

urable architectures as well as other types of loops.

References

[Abdel-Kader, 2008] Rehab F. Abdel-Kader. Particle Swarm Optimization for Constrained Instruction Scheduling. *VLSI Design*, 2008:7, 2008.

[Abielmona, 2009] Rami Abielmona. Reconfigurable Computing Architectures. http://www.site.uottawa.ca/ rabielmo/personal/rc.html, December 2009.

[Acharjee and Goswami, 2009] P. Acharjee and S.K. Goswami. Expert algorithm based on adaptive particle swarm optimization for power flow analysis. *Expert Systems with Applications*, 36(3, Part 1):5151 – 5156, 2009.

[Allan *et al.*, 1995] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[Alsolaim *et al.*, 1999] A. Alsolaim, J. Becker, M.Glesner, and J. Starzyk. A Dynamically Reconfigurable System-on-a-Chip Architecture for Future Mobile Digital Signal Processing. In *European Signal Processing Conf. EUSIPCO2000*, November 1999.

[Alsolaim, 2002] Ahmad M. Alsolaim. *Dynamically Reconfigurable Architecture For Third Generation Mobile Systems.* PhD thesis, Ohio University, August 2002.

[Barr, 1998] Michael Barr. A Reconfigurable Computing Primer. In *Multimedia Systems Design*, pages 44–47, September 1998.

[Beaty, 1994] S.J. Beaty. List scheduling: alone, with foresight, and with lookahead. In *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*, pages 343–347, May 1994.

[Becker *et al.*, 1998] J. Becker, R. Hartenstein, M. Herz, and U. Nageldinger. Parallelization in co-compilation for configurable accelerators-a host/accelerator partitioning compilation method. In *Design Automation Conference 1998. Proceedings of the ASP-DAC '98. Asia and South Pacific*, pages 23–33, Feb 1998.

[Becker *et al.*, 2000] Jürgen Becker, Thilo Pionteck, and Manfred Glesner. *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, volume 1896/2000, chapter DReAM : A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications, pages 312–321. Springer Berlin / Heidelberg, 2000.

[Berekovic *et al.*, 2006] M. Berekovic, A. Kanstein, and B. Mei. Mapping MPEG Video Decoders on the ADRES Reconfigurable Array Processor for next generation multi-mode mobile terminals. In *Proeedings of GSPX 2006: TV to Mobile*, Amsterdam, Netherlands, March 2006.

[Chang and Choi, 2008] Kyungwook Chang and Kiyoung Choi. Mapping control intensive kernels onto coarse-grained reconfigurable array architecture. *SoC Design Conference, 2008. ISOCC '08. International*, 01:I–362 –I–365, nov. 2008.

[Chen and Sheu, 2009] Chang-Huang Chen and Jia-Shing Sheu. Simple paricle swarm optimization. In *Machine Learning and Cybernetics, 2009 International Conference on*, volume 1, pages 460–466, July 2009.

[Ching and Keshab, 1995] Wang Ching, Yi and Parhi Keshab, K. Resource-constrained loop list scheduler for DSP algorithms. *The Journal of VLSI Signal Processing*, 11(1-2):75 -96, October 1995.

[Cho *et al.*, 2007] Doosan Cho, Ravi Ayyagari, Gang-Ryung Uh, and Yunheung Paek. Preprocessing strategy for effective modulo scheduling on multi-issue digital signal processors, 2007.

[Cormen *et al.*, 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 3 edition, 2009. ISBN 0-262-03384-8.

[Davis, 2010] Lawrence "David" Davis. Genetic Algorithms and their applications. http://www.informatics.indiana.edu/fil/CAS/PPT/Davis/, January 2010.

[Dijkstra, 1959] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Dimitroulakos *et al.*, 2007] Grigoris Dimitroulakos, Michalis D. Galanis, and Costas E. Goutis. Design space exploration of an optimized compiler approach for a generic reconfigurable array architecture. *Journal of Supercomputing*, 40(2):127–157, 2007.

[Dimitroulakos *et al.*, 2009] Grigorios Dimitroulakos, Nikos Kostaras, Michalis D. Galanis, and Costas E. Goutis. Compiler assisted architectural exploration framework for coarse grained reconfigurable arrays. *Journal of Supercomputing*, 48(2):115–151, 2009.

[Dorigo *et al.*, 1996] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, Feb 1996.

[Dorigo *et al.*, 2006] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, Nov. 2006.

[Dréo *et al.*, 2006] J Dréo, A Pétrowski, P Siarry, and E Taillard. *Metaheuristics for Hard Optimization*, chapter Ant Colony Algorithms,Simulated Annealing, Tabu Search, pages 123–150. Springer Berlin Heidelberg, January 2006.

[Ebeling *et al.*, 1995] C. Ebeling, L. McMurchie, S.A. Hauck, and S. Burns. Placement and routing tools for the Triptych FPGA. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 3(4):473–482, Dec 1995.

[Ebeling *et al.*, 1997] C. Ebeling, D.C. Cronquist, P. Franklin, J. Secosky, and S.G. Berg. Mapping applications to the RaPiD configurable architecture. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 106–115, Apr 1997.

[Ebeling, 2002] Carl Ebeling. The General Rapid Architecture Description. Technical report, Department of Computer Science and Engineering University of Washington, 2002. UW CSE Technical Report UW-CSE-02-06-02.

[Elmohamed *et al.*, 1998] Saleh Elmohamed, Geoffrey Fox, and Paul Coddington. A Comparison of Annealing Techniques for Academic Course Scheduling. In *Proceedings of the 2nd International Conference on the Practice and Theory of Automated Timetabling*, pages 146–166, Syracuse, NY, USA, April 1998.

[eun Lee *et al.*, 2004] Jong eun Lee, Yunjin Kim, Jinyong Jung, Shinwon Kang, and Kiyoung Choi. Reconfigurable alu array architecture with conditional execution. In *International SoC Design Conference*, 2004.

[Fang, 2000] Min Fang. Layout Optimization for Point-to-Multi-point Wireless Optical Networks via Simulated Annealing & Genetic Algorithm. Technical report, University of Bridgeport,Bridgeport, CT, 2000.

[Gnanaolivu *et al.*, 2010a] R. Gnanaolivu, T.S Norvell, and R. Venkatesan. Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In *Soft Computing and Pattern Recognition , 2010. SoCPaR 2010. International Conference on*, pages 145–151, Paris, France, December 2010. IEEE.

[Gnanaolivu *et al.*, 2010b] R. Gnanaolivu, T.S Norvell, and R. Venkatesan. Modulo scheduling for loops with conditional branches on a coarse-grained reconfigurable architectures. In *Proceedings of Newfoundland Electrical and Computer Engineering Conference (NECEC 2010)*, St. JohnŠs, Newfoundland, November 2010. Awarded GOLD best paper.

[Gnanaolivu *et al.*, 2011a] R. Gnanaolivu, T.S. Norvell, and R. Venkatesan. Analysis of inner-loop mapping onto coarse-grained reconfigurable architectures using hy-

brid particle swarm optimization. *International Journal of Collective Intelligence (IJOCI)*, 2:17–35, 2011.

[Gnanaolivu *et al.*, 2011b] R. Gnanaolivu, T.S Norvell, and R. Venkatesan. Pipelining inter-iteration dependence in loops onto cgras using recurrence-aware priority schemes. In *Proceedings of Newfoundland Electrical and Computer Engineering Conference (NECEC 2011)*, St. JohnŠs, Newfoundland, November 2011.

[Goldstein *et al.*, 2000] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor. PipeRench: a reconfigurable architecture and compiler. *Computer*, 33(4):70–77, Apr 2000.

[Grundy and Stacey, 2008] Ian Grundy and Andrew Stacey. Particle swarm optimization with combined mutation and hill climbing. *Complexity International*, 12, October 2008.

[Guattery and Guattery, 1997] Stephen Guattery and Stephen Guattery. Graph Embedding Techniques For Bounding Condition Numbers Of Incomplete Factor Preconditioners. Technical report, ICASE, NASA Langley Research, 1997.

[Guo, 2006] Yuanqing Guo. *Mapping Applications to a Coarse-Grained Reconfigurable Architecture.* PhD thesis, University of Twente, Netherlands, September 2006.

[Hartenstein *et al.*, 2000] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Kressarray Xplorer: a new CAD environment to optimize reconfigurable datapath array architectures. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pages 163–168, 2000.

[Hartenstein, 2001] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 564–570, New York, NY, USA, 2001. ACM Press.

[Hatanaka and Bagherzadeh, 2007] A. Hatanaka and N. Bagherzadeh. A Modulo Scheduling Algorithm for a Coarse-Grain Reconfigurable Array Template. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[Heath, 1997] Lenwood S. Heath. Graph embeddings and simplicial maps. *Theory of Computing Systems*, 30(1):51–65, 1997.

[Heysters and Smit, 2003] P.M. Heysters and G.J.M. Smit. Mapping of DSP algorithms on the MONTIUM architecture. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 6, April 2003.

[Hu, 2009] Xiaohui Hu. Particle Swarm Optimization. http://www.swarmintelligence.org/index.php, December 2009.

[Huff, 1993] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *In Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, 1993.

[IMEC, 2009] IMEC. ADRES Architecture. http://www2.imec.be, December 2009.

[Kennedy and Eberhart, 1995] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, Nov/Dec 1995.

[Kwok and Ahmad, 1999] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[Lam, 1988] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta,Georgia, June 1988.

[Lee *et al.*, 2010] Ganghee Lee, Kyungwook Chang, and Kiyoung Choi. Automatic mapping of control-intensive kernels onto coarse-grained reconfigurable array architecture with speculative execution. *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –4, apr. 2010.

[Levi and Luccio, 1971] G. Levi and F. Luccio. A weighted graph embedding technique and its application to automatic circuit layout. *Calcolo*, 8(1-2):49–60, March 1971.

[Llosa *et al.*, 1995] J. Llosa, M. Valero, E. Ayguade, and A. Gonzalez. Hypernode reduction modulo scheduling. In *Microarchitecture, 1995. Proceedings of the 28th Annual International Symposium on*, pages 350–360, Nov-1 Dec 1995.

[Llosa *et al.*, 1996] Josep Llosa, Antonio González, Eduard Ayguadé, and Mateo Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *In Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, pages 80–86, Boston, MA, October 1996. IEEE Computer Society Press.

[Llosa *et al.*, 2001] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *Computers, IEEE Transactions on*, 50(3):234–249, Mar 2001.

[Marshall *et al.*, 1999] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143, New York, NY, USA, 1999. ACM.

[Mei *et al.*, 2002] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pages 166–173, Dec. 2002.

[Mei *et al.*, 2003a] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *Computers and Digital Techniques, IEE Proceedings*, 150(5):255–261, Sept. 2003.

[Mei *et al.*, 2003b] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable archi-

tectures using modulo scheduling. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 296–301, 2003.

[Mei *et al.*, 2005a] Bingfeng Mei, Andy Lambrechts, Diederik Verkest, Jean-Yves Mignolet, and Rudy Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design andTest of Computers*, 22(2):90–101, 2005.

[Mei *et al.*, 2005b] Bingfeng Mei, F.-J. Veredas, and B. Masschelein. Mapping an h.264/avc decoder onto the adres reconfigurable architecture. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 622 – 625, aug. 2005.

[Mei *et al.*, 2005c] Bingfeng Mei, F.J. Veredas, and B. Masschelein. Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 622–625, August 2005.

[Milicev and Jovanovic, 1998] Dragan Milicev and Zoran Jovanovic. Predicated software pipelining technique for loops with conditions. *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 176 –180, mar. 1998.

[Newsome and Song, 2003] James Newsome and Dawn Song. GEM: Graph EMbedding for Routing and Data-Centric Storage in Sensor Networks without Geographic Information. In *Proceedings of the First ACM Conference on Embedded Network Sensor Systems*, pages 76–88. ACM Press, November 2003.

[Nonsiri and Supratid, 2008] S. Nonsiri and S. Supratid. Modifying ant colony optimization. *Soft Computing in Industrial Applications, 2008. SMCia '08. IEEE Conference on*, pages 95 -100, jun. 2008.

[Oh *et al.*, 2009] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. *SIGPLAN Not.*, 44(7):21–30, 2009.

[Park *et al.*, 2006] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke. Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures. In *in CASES Š06: Proceedings of the 2006 international conference on Compilers, architecture*, pages 136–146. ACM Press, 2006.

[Park *et al.*, 2008] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 166–176, New York, NY, USA, 2008. ACM.

[Peters and Square, 2011] Tim Peters and Kendall Square. Livermore loops coded in c. http://www.netlib.org/benchmark/livermorec, November 2011.

[Rau, 1994] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM.

[Shekhawat *et al.*, 2009] Anirudh Shekhawat, Pratik Poddar, and Dinesh Boswal. Ant Colony Optimization Algorithms : Introduction and Beyond. http://www.cse.iitb.ac.in/ pratik/projectreports/aco.pdf, 2009.

[Singh *et al.*, 2000a] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465–481, May 2000.

[Singh *et al.*, 2000b] H. Singh, Guangming Lu, Ming-Hau Lee, E. Filho, R. Maestre, F. Kurdahi, and N. Bagherzadeh. MorphoSys: case study of a reconfigurable computing system targeting multimedia applications. In *Design Automation Conference, 2000. Proceedings 2000. 37th*, pages 573–578, 2000.

[Smelyanskiy *et al.*, 2004] Mikhail Smelyanskiy, Scott Mahlke, and Edward S. Davidson. Probabilistic predicate-aware modulo scheduling. *Code Generation and Optimization. 2004. CGO 2004. International Symposium on*, pages 151 – 162, mar. 2004.

[Smit *et al.*, 2007] Gerard J. M. Smit, André B. J. Kokkeler, Pascal T. Wolkotte, Philip K. F. Hölzenspies, Marcel D. van de Burgwal, and Paul M. Heysters. The Chameleon architecture for streaming dsp applications. *EURASIP Journal of Embedded System*, 2007(1):1, 2007.

[Song *et al.*, 2008] Xiaoyu Song, Chunguang Chang, and Yang Cao. New particle swarm algorithm for job shop scheduling problems. In *Intelligent Control and*

Automation, 2008. WCICA 2008. 7th World Congress on, pages 3996–4001, June 2008.

[Sánchez and González, 2001] J. Sánchez and A. González. Clustered Modulo Scheduling in a VLIW Architecture with Distributed Cache. *Journal on Instruction Level Parallelism (JILP)*, 3, October 2001.

[T.Chiang *et al.*, 2006] T.Chiang, P. Chang, and Y.Huang. Multi-Processor Tasks with Resource and Timing Constraints Using Particle Swarm Optimization. *IJCSNS International Journal of Computer Science and Network Security*, 6(4), April 2006.

[Teifel and Manohar, 2004] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *In Proceedings of International Symposium on Asynchronous Circuits and Systems*, pages 17–27, 2004.

[Texas A&M University-Kingsville, 2009] Texas A&M University-Kingsville. Lattice LPC analysis filter. http://www.engineer.tamuk.edu/SPark/Analysis-Synthesis.htm, December 2009.

[Texas Instruments. inc, 2009] Texas Instruments. inc. DSP Benchmarks. http://dspvillage.ti.com, May 2009.

[Thenorio, 2010] Alexandre Weffort Thenorio. Genetic Algorithms. http://www.cs.chalmers.se, January 2010.

[Todman *et al.*, 2005] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design

methods. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):193–207, Mar 2005.

[Tuhin and Norvell, 2008] M. Tuhin and T.S. Norvell. Compiling parallel applications to coarse-grained reconfigurable architectures. In *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*, pages 001723–001728, May 2008.

[Tuhin, 2007] Mohammed Ashraful Alam Tuhin. Compiling Parallel Applications to Coarse-Grained Reconfigurable Architectures. Master's thesis, Memorial University of Newfoundland, St. John's Newfoundland and Labrador, Canada, July 2007.

[University of California, 2009] University of California. Morphosys Architecture. http://www.eng.uci.edu/morphosys, December 2009.

[University of Patras, 2009] University of Patras. VLSI design. http://www.vlsi.ee.upatras.gr, December 2009.

[Uysal and Bulkan, 2008] Ozgur Uysal and Serol Bulkan. Comparison of Genetic Algorithm and Particle Swarm Optimization for Bicriteria Permutation Flowshop Scheduling Problem. *International Journal of Computational Intelligence Research*, 4(2):159Ű175, 2008.

[Vassiliadis and Soudris, 2007a] Stamatis Vassiliadis and Dimitrios Soudris. *Fine- and Coarse-Grain Reconfigurable Computing.* Springer Netherlands, 2007.

[Vassiliadis and Soudris, 2007b] Stamatis Vassiliadis and Dimitrios Soudris. *Fine- and Coarse-Grain Reconfigurable Computing*, chapter 6, pages 255–297. Springer Netherlands, 2007.

[Wang *et al.*, 2001] T. Y. Wang, K. B. Wu, and Y. W. Liu. A simulated annealing algorithm for facility layout problems under variable demand in cellular manufacturing systems. *Computers in Industry*, 46(2):181 – 188, 2001.

[Wang *et al.*, 2007] Xiangyang Wang, Jie Yang, Xiaolong Teng, Weijun Xia, and Richard Jensen. Feature selection based on rough sets and particle swarm optimization. *Pattern Recognition Letters*, 28(4):459 – 471, 2007.

[Warter *et al.*, 1992] Nancy J. Warter, Grant E. Haab, Grant E. Haab, and Krishna Subramanian. Enhanced Modulo Scheduling For Loops With Conditional Branches. In *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on*, pages 170–179, Dec 1992.

[Warter *et al.*, 1993] Nancy J. Warter, Daniel M. Lavery, and Wen mei W. Hwu. The benefit of predicated execution for software pipelining. *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, i:497 – 506 vol.1, jan. 1993.

[Wu, 2011] Shuang Wu. Dataflow synthesis and verification for parallel object-oriented programming languages. Master's thesis, Memorial University of Newfoundland, March 2011.

[Xiaoyu Song and Cao, 2008] Chunguang Chang Xiaoyu Song and Yang Cao. New Particle Swarm Algorithm for Job Shop Scheduling. In *Proceedings of the 7th World Congress on Intelligent Control and Automation Chongqing, China*, June 2008.

[Zalamea *et al.*, 2001] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pages 160–169, Dec. 2001.

[Zalamea *et al.*, 2004] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Register constrained modulo scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 15(5):417–430, May 2004.

Appendix A

HARPOL code for inhouse ifthen-else benchmarks

A.0 ifthen-else benchmark -one condition

```
(class ifthenex

  constructor()

  private obj a :=3

  private obj b := 0

  private obj c := 1

  private obj d := 1

  private obj e := 1

   (thread
```

```
        (if a%2=0 then
            b:=a-c
            e:=b+d
        else
            b:=a+c
            d:=c+e
        if)
        a:=b
    thread)
class)
obj obj1 := new ifthenex()
```

A.1 ifthen-else benchmark -two conditions

```
(class ifthenex
  constructor()
  private obj a := 3
  private obj b := 0
```

```
private obj c := 1

private obj d := 1

private obj e := 1

 (thread

     (if a%2=0 then

          (if b<0 then

               b:=a-c

               e:=b+d

          else

               b:=a+c

               d:=c+e

          if)

     else

          c:=c+1

     if)

     a:=b

d:=e

thread)
```

```
class)

obj obj1 := new ifthenex()
```

A.2 HARPOL code ifthen-else benchmark -three conditions

```
(class ifthenex

   constructor()

   private obj a := 3

   private obj b := 0

   private obj c := 2

   private obj d := 1

   private obj e := 1

    (thread

        e:=e*3

        (if a%2=0 then

            c:=c+1

            (if b<0 then

                b:=a-c
```

```
                    (if e>d then
                        d:=d+b
                        e:=c+d
                    else
                        b:=a+c
                    if)
                if)
        else
                c:=c-1
                d:=c+e
                e:=e/2
        if)
a:=d
b:=e
thread)
class)
obj obj1 := new ifthenex()
```